BASIS TOKEN CONSISTENCY
Extending and Evaluating a Novel Web Consistency Algorithm *

Adam D. Bradley and Azer Bestavros
Computer Science Department, Boston University
{artdodge,best}@cs.bu.edu

June 10, 2002

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 currently standardized web protocols is drawing greater attention. Toward this end, we propose 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 strongly consistent views of content retrieved from supporting servers. We discuss this protocol and its extensions, and compare the performance of BTC with the traditional TTL (Time To Live) algorithm under a range of synthetic workloads.

1 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 near web clients. This has taken a number of forms: Basic caching, varieties of prefetching, and more recently, Content Distribution Networks (CDNs). What has become increasingly clear as these mechanisms have been explored is that the principle difficulty for effective content reuse is not limited cache capacity [6, 25], but managing cache content in a way that is consistent with the applications the web supports. Particularly, one goal is providing clients with a *recent* view of the state of the application (*i.e.*, information that is not too old); another is providing clients with a *self-consistent* view of the application's state as it changes (*i.e.*, once the client has been told something has happened, that client should never be told anything to the contrary). While current web protocols [12] address the first using expiry mechanisms, there is only very limited support for the second.

In this paper we discuss BTC (Basis Token Consistency), an extension to the HTTP protocol which employs concepts from prior web cache consistency and coherence research [7, 15, 26, 17] and from modern content management systems [14, 9, 8]. When when implemented at the server, BTC allows any supporting agent to maintain a view consistent [13] cache without requiring any non-trivial cooperation from intermediaries. We discuss the theoretical and practical strengths of BTC, as well as several useful extensions to it. We then discuss a set of models and simulations we constructed to examine the performance and behavior of our algorithm under a wide range of workloads, and compare it with the conventional TTL algorithm. Our definitions of consistency and coherence, the basic overview of BTC's syntax and algorithm, and portions of our simulation results will also appear in [4].

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

1.1 Consistency and Coherence within a Web-like Framework

The web does not behave like a distributed file system (DFS) or distributed shared memory (DSM) system; among the distinctions are: (1) The lack of a "write" semantic in common use - while the HTTP protocol does include a "PUT" event which is in some ways comparable to a write, it is rarely used; the most common write-like operation is "POST" which can have completely arbitrary semantics and scope. This generality implies, in the general case, an inability to "batch" user-induced updates. (2) The complexity of addressing particular content - URIs or "web addresses" do not in fact address units of content *per se*, but rather address generic objects ("resources") which produce content using completely opaque processes. (3) The absence of any protocol-layer persistent state or notion of "transactions" to identify related, batched, or macro-operations. These issues are further illuminated by Mogul in [20].

In a DSM of DFS world, the mapping from write events to eventual changes in the "canonical" system state is clearly defined; in the web, non-safe [12, §9.1.1] requests from users can have arbitrary application-defined semantics with arbitrary scopes of affect completely unknowable from the parameters of a request, or even from the properties of a response. For this reason, the definitions of consistency and coherence used in the DFS/DSM literature do not fit the needs of systems like the web; instead, we use definitions more akin to those in the distributed database literature.

Consistency For our purposes, cache *consistency* refers to a property of the responses produced 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. Said another way, a consistent cache provides its clients with *non-decreasing* views of the server's application state. This is also the definition of consistency used in [14, 8].

This definition is distinct from traditional definitions of web consistency [7] in that it recognizes that web pages are not atoms of information; rather, consistency is with respect to the set of information sources used by the web server's application logic to construct the pages. These sources we call *origin data*.

Coherence In the web, a *coherence* mechanism is one which propagates updates to entities through the caching network such that all clients interested in entities effected by those updates eventually see their results. Said another way, a coherence mechanism deals with the *recency* of cached content.

Note that a cache can be *consistent* without being *recent*, and visa-versa. One model for coherence, *immediate coherence* (in which a cache never serves a response older than that which the origin server would provide itself at that point in time), provides consistency as a side-effect when it can be guaranteed correct.

2 Basis Token Consistency

We have devised a web cache consistency protocol we call "Basis Token Consistency" (BTC) [3]. When implementing BTC, an origin server annotates all response with enough information to allow any cache interested in maintaining consistency to detect if other responses it currently holds are rendered obsolete by information included in that response, or to identify responses it receives from other caches as having already been rendered obsolete.

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

Figure 1: The `Cache-Consistent` HTTP Entity Header

To accomplish this, each response the server generates includes a `Cache-Consistent` header with a set of *basis tokens* (opaque identifier strings), each with a *generation number* (in hexadecimal); the augmented-BNF for this header is presented in Figure 1. An example header produced by the server `www.cs.bu.edu` could look something like this:

```
Cache-Consistent:  studentdb;4e9,
     db2row@bu.edu;7a,
     form01@cs.bu.edu;2
```

Each basis token represents some dynamic source of information in the underlying application; each such source is called an *origin datum*. All responses which depend upon a particular origin datum will include its basis token (`cctoken`) in their `Cache-Consistent` header, and whenever a datum is changed, its generation number (`ccgeneration`) is incremented such that all responses produced using it in the future will reflect that new generation number. The aggregation of all such tokens and their generation numbers can be thought of as a *vector clock* [11, 18] on the state of the system's underlying origin data.

Caches implementing BTC keep an index of cached responses keyed on basis tokens. Whenever a new response arrives, each of its tokens' generation numbers are compared with the cache's "current" generation numbers for those same tokens. If the generation numbers match, no further action is taken. If the new generation number is greater, all entities dependent upon the older generation of that token are marked as invalid and the "current" generation number is updated to the new value. If the new generation number is less than the current value, then the new response itself is stale and inconsistent (most likely produced by an inconsistent upstream cache), so the request should be repeated using an end-to-end reload.

As with cookies, basis tokens are always scoped to particular DNS hostnames or domains; this allows some origin data to be used by multiple servers within some broader administrative domain. The default token scope if none is specified is the `Host` identifier used in the client's request; this can be overridden by a `cctokenscope` value. The scope string (whether the implicit default value or given explicitly) is considered part of the token's identifier for the purpose of identifying matching tokens. To frustrate certain kinds of denial-of-service attacks, an entity may only scope tokens to one of its own fully-qualified domain names or to a superdomain (suffix) thereof. Tokens violating this rule must be discarded by all downstream clients. The idea is that token scopes correspond (at least nominally) with administrative domains; attempts to scope outside of your own administrative hierarchy (e.g., the server www.cs.unca.edu trying to provide a basis token for `@eng.unca.edu`) will be rejected outright, and scoping a token "above" your administrative control (www.bu.edu setting a token for `@edu`) would allow other hosts outside of your administrative control to "hijack" your basis tokens legally, making such an action inherently undesirable.

2.1 Features

Some of the properties which distinguish BTC from conventional [7], recent [26, 15, 17] and current [22, 23] web consistency proposals include: (1) Strong point-to-point consistency does not rely on the cooperation of intermediaries; (2) Invalidations are automatically aggregated; (3) Aggregation is independent of HTTP namespaces; (4) Invalidation is driven by the application, not by heuristics; (5) All data is lazily delivered; (6) All transmitted data pertains to the entity it annotates; (7) No per-client state is required at the server or intermediaries.

Strong point-to-point consistency To a non-BTC intermediary, the `Cache-Consistent` header is just another cachable metadata header which it will not remove or alter. Consider a situation where a BTC origin server and a BTC cache are communicating via a non-BTC cache which has itself acquired inconsistent contents; for any given basis token, once the downstream BTC cache has seen a particular generation number it sets a new "low water mark" for that token's generation numbers, meaning that any entity bearing a lower generation number can be immediately identified as the product of an inconsistent cache and discarded. Thus, the detection of inconsistent intermediaries is trivial; remedying the situation is also straightforward, because HTTP allows a client (the BTC cache) to initiate a forced end-to-end reload which will get the most recent version of the response from the origin server. Consequently, BTC yields correct results even without any special cooperation from intermediaries.

Invalidations automatically aggregated Aggregation of invalidations is implied by the fact that single basis tokens are shared by multiple resources. Thus, the scope of aggregation precisely corresponds with the granularity of the origin data; rather than grouping responses using coarse volumes, we are able to only invalidate those cached response which are in fact affected by an update to the server's origin data. This aggregation is provided automatically by the mechanism itself and the underlying object model of the server; it does not require the construction of "classes" or other explicit object groupings as would [26] or [22].

Aggregation independent of HTTP namespaces The namespace of BTC tokens is independent of URIs and almost

all of the HTTP parameter space[1]; the one exception is the coupling of the `Host` request header with token scoping. Thus, there is no need for artifacts like common URI substrings to pollute the application namespace in order to identify aggregation classes.

Invalidation driven by application Rather than taking a predictive approach, BTC is directly coupled with the behavior of the underlying application at the server; thus, all invalidations in caches are caused by real events, not heuristics or adaptive estimations of server actions, and are thus "productive" in that an already-consistent cached copy will never be evicted by the BTC algorithm unless the server chooses to control consistency at a coarse granularity which intrinsically forces such unproductive invalidations.

Lazy delivery Many consistency protocols are *push* oriented, in that they require some way for servers to contact their clients asynchronously and notify them of invalidation events. While such approaches certainly have their benefits, they must contend with several problems: First, the internet as it exists today is not symmetric; firewalls and IP masquerading are realities that prevent push-like communication channels from working with caches which live within protected/masqueraded domains. Second, such a scheme relies upon the entire caching network supporting it in order to properly disseminate invalidation messages; client downstream from a non-supporting cache are left in the dark with respect to invalidations. Third, such techniques need to be mindful of potential scalability problems that arise from the need to have a list of clients to contact with update notifications. By contrast, BTC communication all takes place in the context of normal client-initiated HTTP requests, not unlike the piggyback invalidation mechanism of [15].

Data pertains to entity it annotates Unlike other lazy delivery methods, responses are only annotated with enough data to keep caches consistent *with respect to that particular response*. In a sense this limits BTC in that it is prevented from affecting any evictions which are not consistency-related with a user's page discovery/browsing path; pedantically speaking, however, that is an issue of coherence (eventual update of unrelated pages), not consistency (temporally non-decreasing view of the state of server resources). At the same time, this spares us a complication in the presence of non-implementing intermediaries: when a mechanism annotates normal responses with arbitrary cache eviction messages, what is produced are single "overlaid" responses which contain two completely orthogonal message components. Since a non-implementing cache is unable to separate these two, it caches them together, thus transmitting a cache invalidation message to all of its downstream clients whenever it happens to reuse that cached response.[2] This property also greatly simplifies the implementation of BTC at the server, since the scope of backing information that need be examined to produce a response is not enlarged (as it would, for example, in order to consult a queue of eviction or update events).

No per-client state at server or cache BTC requires no notion of leases or subscriptions; no client or proxy needs to maintain any per-client information for the system to behave correctly. While most techniques requiring per-client state include mechanisms for managing this scalability problem [16, 23, 15], it is simply a non-issue for ours.

2.2 Requirements and Limitations

Server Application Support Unlike other approaches, BTC will not work effectively without support from the applications "behind" the web server. The basis tokens are essentially offering a window into the state of databases, files, and other resources which those applications normally segregate from the outside world.

Increased Cache State The tracking of token generation numbers and token-to-entity relationships places an additional storage burden upon caches. However, we argue that this burden is reasonably small in space and algorithmic complexity. While an entity can be affiliated with arbitrarily many tokens, in practice the number will tend to be fairly small, and reuse of some tokens will tend to be very high.[3] As such, we expect the number of tokens in the index to grow roughly linearly with the number of URIs; more precisely, we expect that when a server is first visited, there will be a spike as the "popular" tokens are introduced, and from that point on we would expect roughly linear discovery, perhaps tempered by an asymptotic fall-off as the user approaches "full coverage" of the site's complete set of basis tokens. Since scoped tokens occupy a flat global namespace, responses can be indexed on their tokens using the same data structure which indexes URIs[4], implying a constant-factor[5] increase in index storage space and update complexity.

Large Stack Distances The BTC algorithm's ability to guarantee the self-consistency of an output stream *beyond the cached lifespans of responses derived from a particular token* is bounded by the ability to retain a long-term history of token generation numbers seen in the past but not presently represented in-cache. When the span (stack distance) of accesses to consistency-related resources exceeds the capacity of this history mechanism, inconsistencies introduced upstream can go unnoticed. It is hoped that such history mechanisms will have adequate capacity such that traditional expiration or other coherence mechanisms will kick in before this problem can manifest.

Coherence v. Consistency BTC is purely a consistency (as we have defined it) mechanism; it is completely orthogonal to the issue of cache coherence (as we have defined it). BTC will not identify a perfect 1-year-old snapshot of a server as "inconsistent", because it is not; that snapshot may be stale, but it is not inconsistent. This separability is actually a desirable property in several ways; consider the possibilities for disconnected/poorly-connected operation [13], or the flexibility this offers us in selecting different coherence mechanisms based upon the needs of the underlying application and the available communication infrastructure (symmetric v. asymmetric reachability, for example).

[1] e.g., the `Accept` header family

[2] This problem can can also be dealt with by explicitly versioning responses and invalidation messages, allowing further-downstream caches to correctly discriminate stale invalidation messages.

[3] It would make intuitive sense for their popularity to follow a Zipf-like distribution [27]; while we lack evidence from any reasonably large sampling of "real" sites, the data presented in Figure 11 of [8] lends some credence to this theory.

[4] It is true that a single token will usually point to several cached responses; however, in HTTP/1.1 caches the same can be said of URIs, since a cache may hold multiple "variants" of a particular resource simultaneously [12].

[5] That factor is roughly the ratio of unique basis tokens to cached responses.

3 Enhancements to BTC

We have defined several extensions to the basic BTC mechanisms and algorithm discussed above.

A BTC-Based Coherence Mechanism We can augment BTC with a lazy-delivery coherence system that will allow arbitrary responses sent to a cache to invalidate sets of entities based upon arbitrary basis tokens. Doing so requires either a return to per-client state tracking (the "directory-based coherence" model - we need a mapping from basis tokens to clients consistent with their "current" versions) or a more pessimistic "broadcast" approach (attach coherence messages to every outgoing response, leading to tremendous redundancy); the latter is only reasonable for promoting coherence based upon a relatively small number of tokens, and the former has the inherent scalability issues discussed above.

The premise is fundamentally the same as that of the consistency mechanism: responses can be annotated with a set of basis token/generation number pairs. These are not associated with the entity carried by the response, but are rather interpreted as *global* and *advisory*; in principle, they are used just as the token;generation pairs of the `Cache-Consistent` header are used to identify inconsistent cache entries and invalidate them, with the exception that they do not force equality but simply advise of a potential lower bound for a particular token's generation number. This means that if a message includes a coherence token with a "stale" version number, that particular token;generation pair can be disregarded, while the rest of the message is evaluated independently. Said another way, a coherence token being stale *does not* imply that the response carrying it is itself inconsistent; this is why we call coherence tokens *advisory*, as opposed to consistency tokens being *compulsory* (they must match to provide correctness).

As a curiosity, we note that in the extreme a server is able to force lazy-immediate coherence (a cache is made aware of all of its currently-stale entities whenever it next contacts their corresponding server) by including in the next message to that cache the `Cache-Coherent` header with all of its current basis tokens and generation numbers.

The grammar for `Cache-Coherent` is virtually identical to that of `Cache-Consistent` presented in Figure 1.

Strong End-to-end Consistency While BTC in itself guarantees that any cache-server pair wishing to participate can maintain strong consistency, this guarantee does not extend beyond the furthest downstream BTC cache; we are able to provisionally extend our guaranteed consistency by extending HTTP's `Cache-Control` mechanism with a new parameter, `cc-maxage`. BTC caches give this value primacy over the `Expires` header and the standard `max-age` and `s-maxage` parameters, while non-BTC caches will ignore it; thus, an entity can be pre-expired using `max-age`, "busting" all non-BTC caches (including the client's), with a special provision allowing only BTC caches to retain and reuse the content.

This feature is provisional in that it will only work correctly if the caching network topology meets certain conditions; this mechanism will fail in certain cases if the caching network is *divergent*. Upstream path divergence is the property of the cache network upstream of a client node such that requests directed to a single logical server can take different (diverging) logical paths through that network. This makes it possible for the response stream seen at the "split point" to be internally inconsistent even if all caches along each of the upstream paths provide internally consistent response streams. To illustrate this, recall that consistency does not imply recency; imagine a user routes her requests randomly to one of two upstream proxy caches (thus, a diverging path). Each of those caches is internally kept consistent by some mechanism, but the coherence mechanism does not guarantee that they will both reflect the state of the server at the same point in time.[6] One cache may have captured a self-consistent "snapshot" of the server's state that is now five minutes old; the other has similarly captured a snapshot that is only one minute old. In the intervening four minutes, a number of changes were made on the server; the user is now in a situation where she can make one request and get a representation of the recent state of the server, and make a subsequent request and get a representation of an older state. This potentially "decreasing" view of the server's state is, by definition, inconsistent, and unless some downstream node is able to identify and rectify this condition, the client will see an inconsistent response stream. While BTC is in fact able to handle this situation properly when the divergence is upstream of the furthest-downstream BTC implementation, if a client can reach a divergence before it reaches a BTC cache then this mechanism is unable to guarantee the consistency of the response stream seen by the client.

Minimal View Consistency Assuming a non-divergent caching network, two techniques could be applied to more aggressively offer view consistency to clients without forcing unnecessary invalidations upon further downstream caches (as may be desirable in a low-bandwidth or intermittent-connectivity environment, one of the motivations for [13]). The first technique, Stateful View-consistent BTC (SVBTC), introduces per-client state at the proxy; the second, Parameterized View-consistent BTC (PVBTC), is per-client stateless but requires clients to provide additional information in their requests.

For either of these view-consistent caching techniques, the shared caches no longer evict stale entities, but rather retain them (along with the basis token generation numbers used to construct them) on the possibility they may be useful in answering future requests from clients who wish to avoid invalidating their own cache entries whenever possible. The idea behind both SVBTC and PVBTC is that the cache is informed of the token generation numbers which are "current" for the client's cache and, if possible, selects responses which share those generation numbers. It is possible for there to be multiple cached responses for a given resource which match the known state for a client; this happens when a response includes a basis token not previously seen by the client. When this occurs, the implementer is free to use whatever selection method best fits her goals; two useful models could be "lexically greatest unknown-token values" (to try to maximize recency) and "token values most widely used in other cached resources" (to try to maximize the cache's ability to use an already-held partial snapshot of the server).

In the SVBTC approach, the cache retains a complete list of the basis token vectors seen by all of its clients, and uses this data to inform its choice of responses. This is taxing upon the cache, but it imposes no additional communication demands downstream, and interacts without problems with stock BTC clients.

In the PVBTC, we borrow the concept of the `A-IM` header from [19]: the client annotates every request it makes to the

[6] Such situations can be minimized, but not completely eliminated, using synchronous coherence mechanisms. Even immediate coherence will not guarantee correctness, since there is no strong relationship between the order in which requests are made to the proxies and the order in which those proxies make their requests to the origin server. See the analyses of inherent race conditions in conventional consistency and coherence schemes presented in [3] for more details.

cache with the subset of its complete basis token vector which may be relevant to that request; every token whose scope value matches[7] the target Host's name must be reported in the request. These tokens are provided using the `Request-Consistent` header, also with a grammar virtually identical to `Cache-Consistent` (presented in Figure 1). PVBTC allows the upstream cache to operate statelessly, but clearly the communication burden can become prohibitive in the presense of a non-trivial number of basis tokens, particularly considering we may be in a limited-bandwidth environment. The stateless operation of the cache also introduces a garbage collection problem; where in SVBTC the cache could decide if none of its clients are interested in a particular version of a response anymore (because its basis token vector isn't a subset of any client's current vector), in PVBTC the cache has no such knowledge and must employ some sort of heuristic approach to decide when evictions should take place.

Relaxing Consistency Demands The notion of bounded staleness as a way of relaxing consistency is widely employed (see $\Delta(t)$ consistency in [23] or staleness bounds in [24] for two examples). What a mechanism like BTC allows us to do is bound *logical* staleness (*i.e.*, staleness with respect to a logical version-identifying value) as opposed to *temporal* staleness (staleness with respect to a clock).

For BTC, we support relaxed consistency by allowing token;generation pairs in the `Cache-Consistent` header to also include an optional pair of *margin* parameter. This extends the BNF for `cctokengeneration` using the rules shown in Figure 2.

```
cctokengeneration =
     cctoken
     '';'' ccgeneration
     [ ccdoesnot ] [ ccwillnot ]
ccdoesnot = ''-'' ccmargin
ccwillnot = ''+'' ccmargin
ccmargin = 1*HEX
```

Figure 2: Extended BNF for Relaxed-Consistency BTC

The `ccdoesnot` parameter carries the semantic of "an entity with this token with a generation number not more than `ccmargin` less than this current one is not invalidated by this response"; the `ccwillnot` parameter denotes "future responses with a generation number not more than `ccmargin` more than this current one do not invalidate this response." A `cctokengeneration` can include both parameters.

A well-designed versioning system can take advantage of this fine-grained control over the margins to impose structure upon the linear generation number space. For example, say a CVS-like two-dimensional version numbering scheme is used internally by the application (1.3, 2.7, 5.0, etc), and that we do not wish for minor-version increments to provoke cache invalidations. Say version 5.0 of a datum is generation number 604 (2c0 hex). Responses derived from subsequent releases are numbered in parallel, with the `ccdoesnot` margin stretching back to the 5.0 generation number; thus, 5.1 would be presented as `2c1-1`, 5.2 `2c2-2`, and so on, until version 6.0 is reached and the `ccdoesnot` parameter is reset to zero (the implied value if the parameter is absent). Similarly, since generations numbers do not need to strictly increment but can increase with any monotonic non-decreasing process[8], the major-version number could be defined to occupy a high-order position (say, the fourth hexadecimal digit); thus, 1.0 would be encoded by a `ccgeneration` of `1000`, 2.0 by `2000`, and so on. Then, any response could include a `ccwillnot` parameter which stops just short of the next major revision generation number; for example, `10e2+f1d`.

Nesting Consistency Issues In the web, users do not actually interact with resources strictly serially, but rather with groups of resources which are co-displayed simultaneously by way of *nesting*; for example, a single web page may embed a number of images, frames, and layers, each being fetched independently at a distinct point in time, each from a different resource address. Making sure the set of co-displayed resources are self-consistent is a difficult problem, since it adds a parallelism requirement to our thus-far very serial definition of consistency.

While the stock BTC algorithm is capable of recognizing an inconsistency among co-displayed documents, it is not capable of ensuring that a self-consistent view can be fetched. One way to do this correctly is to employ ideas presented in the "Minimal View Consistency" extension defined above; if we extend either the stateful or parameterized model all the way from the client (which determines the display-relations among resources) to the server (which maintains an adequate resource revision history), then nested objects can have their served versions controlled by the basis token vector of the embedding objects.

A less heavy-handed approach would be to use relaxed logical consistency for embedded objects. If we model a nested document presentation using a tree, we know that the client's fetching strategy will always be some sort of prefix iteration over that tree (*i.e.* parent "embedder" nodes will always be fetched before child "embeddee" nodes); this means that, following a conventional fetch ordering, an embedded element (if fetched from a consistent cache) will never be stale *with respect to the embedder*, but by the time the tree has been completely fetched some embedder may end up being too old with respect to some embeddee, or nodes in some branch may be stale with respect to nodes in some earlier-fetched parallel branch. If these latter conditions are acceptable, then appropriate `ccwillnot` or `ccdoesnot` parameters should be included with some subset of the entities. When the latter conditions are not permissible (as represented by the change not falling within the acceptable margins), a browser may attempt to reconcile it by re-iterating over the tree doing a postfix re-fetches of elements as needed; of course, these re-fetches could give rise to changes in the structure of the tree, or may themselves retrieve entities which are "too new" with respect to the previously newest entities, so some heuristic bound is needed to prevent this process from looping indefinitely.

Ultimately, however, it is not clear how useful such a mechanism is; an even simpler way to deal with this situation where the internal consistency of a set of co-displayed documents is important is to define the problem away as follows: "Whenever an embedder and an embeddee are consistency-related, the embedder's reference to the embeddee SHOULD explicitly encode the embedder's current version information as a parameter to the embeddee's URI."

[7] *i.e.*, "is equal to or a suffix of"

[8] Timestamps are also a suitable source for generation numbers, and allow us to approximate $\Delta(t)$ consistency [23].

4 Simulation

To illustrate the correctness and performance impacts of BTC, we implemented a server-and-cache simulation which compares the performance and correctness characteristics of several consistency models under a range of workloads.

The BTC algorithm relies upon not just changes of documents at the server itself (of which some general study has been done [10, 21]), but upon the "hidden" events within the server's application logic that provoke those changes. For our simulation we therefore needed to model some sort of driving application; we chose to base this upon the architecture of a simple Content Management System. Our hypothetical Content Management System (or CMS) is inspired by IBM's DUP-based system [14, 9, 8], primarily because its organization and algorithms are representative of current practices in industry and well-described in the research literature.

The concept of a CMS is quite straightforward. The documents that make up a modern content-driven web site are themselves aggregations of a number of components: static markup text (such as that describing the banner at the top of the page, copyright information, etc), user-selected markup (for "skinnable" sites), human-created text content (which is regularly added and possibly edited), and automatically generated database-driven content (stock quotes or tennis match results) are typical components. It is the job of the content management system to integrate these parts in such a way that the site's "authors" can focus upon doing their "part" of making it work (writing articles, changing the "look" of the site, etc) without having to deal with or even be aware of the details of how the other "parts" are handled.

For any given resource, the CMS must know how to assemble pieces (called *fragments*) to produce complete entities. This relationship is codified in an *object dependence graph* (ODG); an ODG is a directed graph with a set of nodes representing resources, each with a set of inbound edges from "objects" it uses to create entities. These objects can themselves have inbound edges from other "objects" to whatever depth is needed to represent the site's organization, complexity, and data model. Such "nesting" objects will tend to be control files describing how to combine the fragments provided by other objects; at the "head" of the graph are the underlying data, which tend to be things like flat files, individual keyed database rows, particular keyed sets of database rows, or whole database tables. A simple example of an ODG is shown in Figure 3.

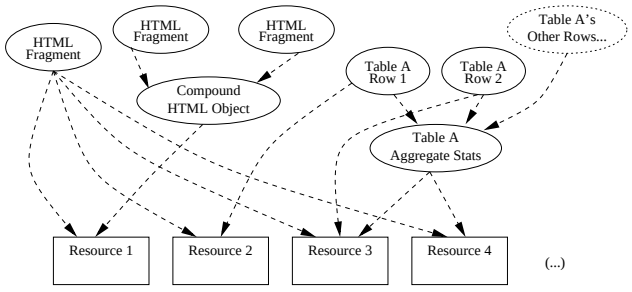

Figure 3: Sample Object Dependence Graph

This graph fully specifies the consistency relationships between underlying origin data (nodes with no inbound edges) and resources, and thus between all entities produced by those resources. When an origin datum is updated, every resource reachable from it is affected, and all of those resources will need to be kept mutually consistent. For our purposes, we require the graph to be acyclic; the cycle semantic of the DUP algorithm [14] can be preserved using a simple transform.

As this graph provides us with all the interdependence information needed to address cache consistency, the decision to represent nodes of the graph with basis tokens is a fairly obvious one. It is trivial to add generation numbers to every node in the graph which are incremented as data are updated. This leaves the question of which of these nodes actually *need* to be represented directly via basis token annotations to entities?

Our first instinct could be to represent all nodes which reach a resource as basis tokens for its entities. While this approach would certainly give us correct results, the graphs of highly involved sites can grow quite large and involve several layers of indirection, which could make the space requirement for transmitting the list unreasonable.[9]

A more attractive solution would be for each resource to only use the nodes of its parentless origin data; while this would also produce correct results, it fails to take advantage of aggregations of clusters of origin data into single "atomic" compound objects which may be present in the graph.

For any given ODG, there exists at least one minimally-sized virtual graph which perfectly captures its original expressive power. As appealing as it may be to find such an optimal graph, however, it is unreasonable to expect the graph to remain static at runtime; nodes and edges will constantly be added, and those patterns which yielded optimal mergings in earlier versions of the graph may cease to be optimal. Given the complexity of maintaining a backward-compatibility graph (and the added cost both in transmission and cache storage space of sending and storing the accompanying tokens) needed should such morphological changes to the optimal graph be necessary, it is far more appealing to use simple heuristic-based on-line approaches which incur the cost of some redundancy but are resilient to those types of changes.

All that remains then is to create a persistent mapping from the selected nodes of the ODG to basis token strings and add persistent version numbering (already present in most managed data sources anyway), and it becomes a trivial programming exercise for the CMS to attach the appropriate `Cache-Consistent` headers to every response it sends. Virtually all of the work is done for us by the CMS's existing architecture; BTC simply adds a small "window" at the publishing phase into the system's internal state.

4.1 Simulation Design

Lacking any thorough statistical and topological study of ODGs found in the wild[10], our model requires a number of assumptions. Rather than claim our simulations are representative, we provided a variety of parameters that would allow us to explore our protocol's performance under a wide variety of potential conditions. The results presented here are illustrative of the qualitative performance properties we observed under varied parameterizations.

We modeled our CMS as a bipartite graph of datum nodes and resource nodes built using two parameters: size (the number of resource and datum nodes) and saturation (the percentage of possible edges in the graph actually present); in our presentation of results below we focus principally upon the "small and dense" setup with 40 resources, 200 datum nodes, and a 50% saturation; we also briefly discuss a broader range of configurations, including small (40 resource, 200 datum), medium (200 resource, 1000 datum), and large (400 resource, 2000 datum) setup sizes, with dense (50%), medium (25%), and light (12.5%) saturations.

[9] For anecdotal evidence, see Figure 10 in [8].

[10] The characterization provided in [8] is illustrative but not necessarily representative.

Each datum is assigned an update process. Results presented here were generated using exponential update processes, with the distributional means themselves being exponentially distributed.

Next, the simulator generates an update sequence consisting of some number of update events (we used 5000 for the small graph size) timestamped according to their update processes.

Resources are assigned popularities according to a Zipf-like popularity distribution [27, 1, 5]; the Zipf parameter, α, could be varied arbitrarily, but all of our experiments use the value 0.7, reflecting approximately the value reported in [1].

A stream of requests with constant inter-arrival times is synthesized, and merged with the stream of update events. The total number or requests is a multiple of the number of update events (1, 20, or 400, labeled *slow*, *medium*, and *fast*). We examine the difference in behaviors for each of these three cases below.

While this is a very simplistic request model in light of current understanding of web workloads [2], given the rather *ad hoc* nature of the backend update model, and given the relationship of many axes of study of request streams (locality of reference, inter-arrival times) with completely unknown properties of the server's backend (locality of data, update process characterizations) we believed it to be unreasonable to build too much complexity and detail into the request stream at this time.

Finally, this combined event stream is fed to the server-cache simulator. This simulator simultaneously maintains a model of the server's state, a number of TTL caches with different fixed time-to-live values, and a set of caches using BTC and the same TTL values (which we call the "Hybrid" algorithm). Upon completing the stream, the simulator reports for each of these the number of requests that missed in the cache for any reason, the number of cached responses which were stale with respect to the origin server, (for TTL caches) the number of responses which were not view consistent with previously served responses, and a normalized "quality" value which indicates what portion of the origin data reflected in the final response stream were still fresh when served.[11] The cache models do not include any notion of capacity misses.

4.2 Simulation Results

All graphs present the time-to-live parameter on the X axis, normalized to the length of the simulation in time[12]. The Y axis is normalized to the total number of requests made in a simulation run.

Figures 4 and 5 show 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 cache or small-population shared cache.

Notice that the TTL algorithm sheds server load (Fig 4 ◯ and Fig 5 □) for moderate time-to-live values, but this is accompanied by a matching falloff in the proportion of fresh responses (Fig 4 ■) and consistent responses (Fig 5 ∗); this is indicative of the large number of "false hits" as TTLs exceed true resource freshness lifespans. The accumulation of poor quality (poor immediacy) is less dramatic; this is not unexpected, as each resource is connected with a large number of origin data, and TTLs tend to expire before too many of these update events can accumulate to render all of a resource's parts

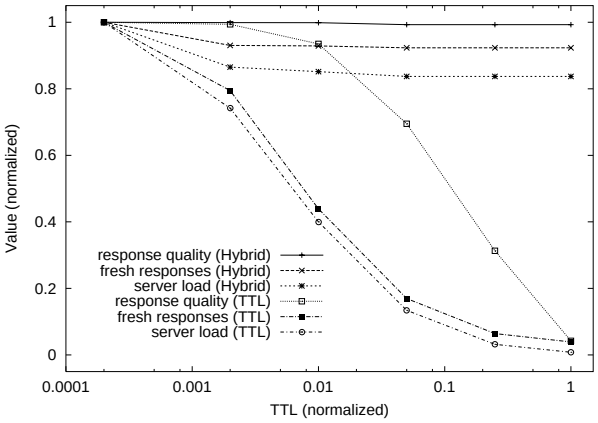

Figure 4: Behavior with Slow Request Stream

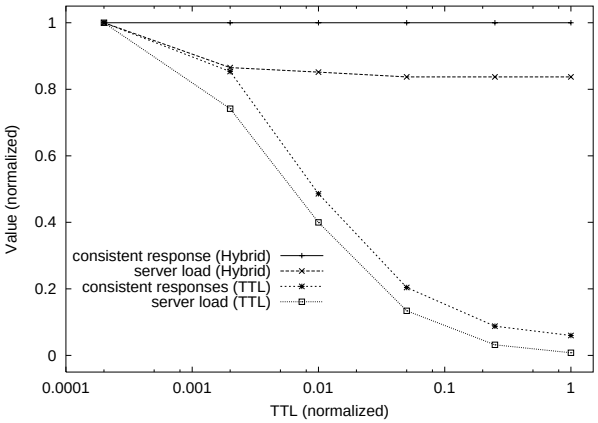

Figure 5: Behavior with Slow Request Stream

out-of-date[13]. TTL's quality value 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 that cause responses to become stale, and the ratio between the average datum update rate and the request rate is constant due to our experimental setup.

At the same time, note that the Hybrid algorithm only allows about 15% of the server's load (Fig 4 ∗ and Fig 5 ×) to be shed. However, its response quality and staleness rates (Fig 4 + and ×) remain very favorable. This is not surprising; more resources are updated in the average unit of time than requests are made, so it is likely that most request for a resources that are consistency-related to other already cached resources will cause those cached entities to be evicted.

Figure 5 shows the tradeoff between consistency and server load under this experimental configuration. 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; notice how under the TTL algorithm server load and consistency (□, ∗) decline in parallel for non-trivial TTL values, while the Hybrid algorithm maintains consistency but at the price of much higher necessary server load (+, ×).

The same setup with a medium-rate request stream shows

[11] Think of the quality value as the continuous counterpart to the all-or-nothing "stale/fresh" per-response flag.

[12] When a resource has a TTL of 1.0 it will never expire within the span of the simulation; thus, "pure BTC" would have the same results as the Hybrid algorithm with a TTL of 1.0

[13] Think of this as an instance of the coupon-collector's problem which works to our advantage.

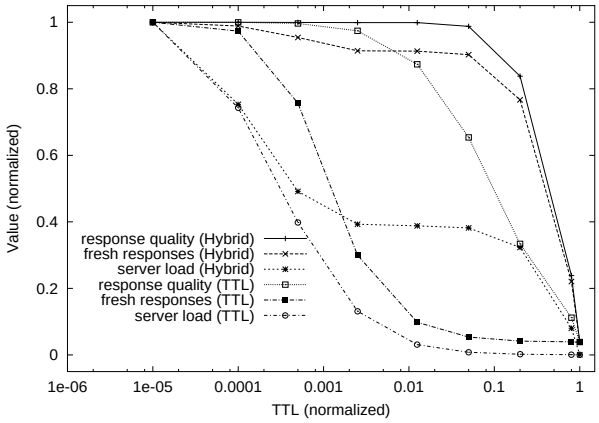

Figure 6: Behavior with Medium-Rate Request Stream

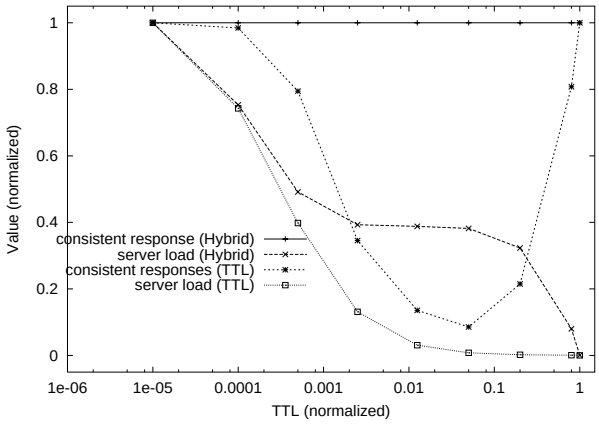

Figure 7: Behavior with Medium-Rate Request Stream

some very interesting behaviors, as illustrated in Figures 6 and 7. 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. We also see a multiplicative shift of the quality of TTL responses relative to the stale/fresh and load metrics similar to that discussed above.

Quality and fresh responses for the Hybrid algorithm both deteriorate quickly under very large TTLs. This makes intuitive sense in light of Figure 7; 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 resources making up that snapshot are long-lived, it stops talking with the server and therefore stops receiving the (lazily delivered) invalidation-provoking new tokens.

This effect (and thus, the need for a more aggressive coherence mechanism) is controlled principally by the interplay of four factors: the number of resources, the average resource update rate, the request rate, and the popularity profile of the resources (in this case, the Zipf parameter). For example, it is hard to get a complete snapshot when the number of resources is particularly large relative to the request rate, when the update rate is high enough with respect to the request rate, or when the Zipf parameter is particularly high (making unpopular resources very rarely be viewed); at the same time, a high Zipf parameter can make partial snapshots longer-lived, precisely because of the rarely-accessed resources (which may have been updated and could therefore "break" the snapshot) are in fact rarely-accessed and therefore seldom have the opportunity to act as snapshot-breakers.

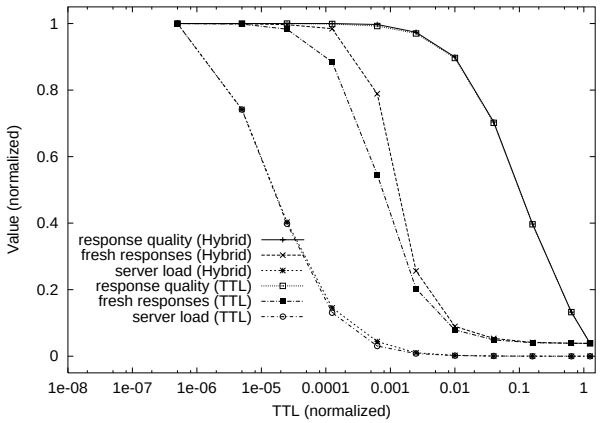

Figure 8: Behavior with Fast Request Stream

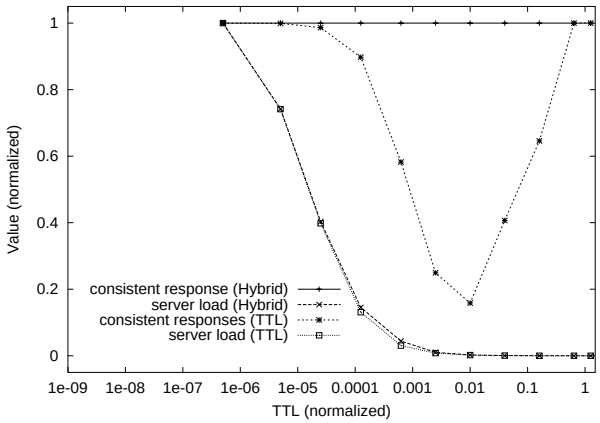

Figure 9: Behavior with Fast Request Stream

Next, we turn our attention to the high request rate run, reflected in Figures 8 and 9. This case further exaggerates the snapshotting issue; we see that load shedding, fresh responses, and response quality are very similar for both algorithms (although the Hybrid algorithm exhibits a significant advantage over TTL in a narrow middle band of TTL values). Again, we see TTL's consistency declining and then recovering for very large TTL values, although the recovery becomes pronounced at lower TTL values (reflecting, again, the increased ability at this much higher request rate to obtain an internally consistent snapshot). The consistency curves for the three request rates are shown together for comparison in Figure 10.

Of course, when choosing a cache consistency algorithm and its parameters (*e.g.* the TTL value), that choice impacts upon a number of independent performance axes: server load, response consistency rate, and response quality or freshness have all been discussed above independently; in Figure

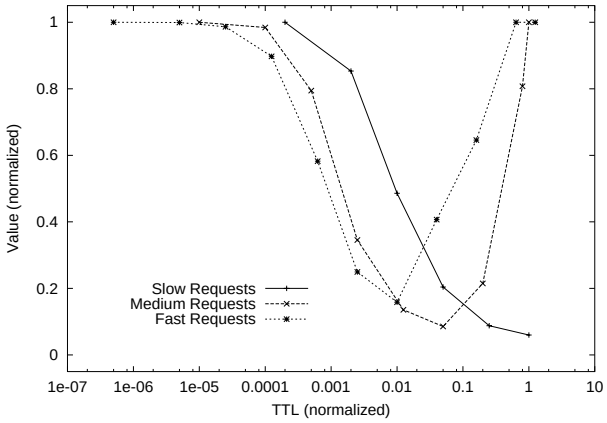

Figure 10: TTL Consistency under Various Request Rates

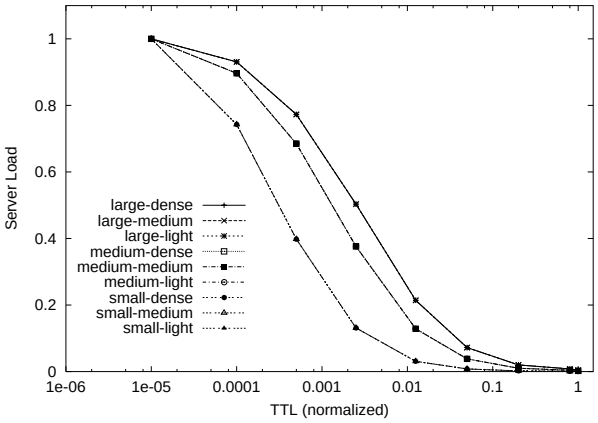

Figure 12: TTL Load Effects for All Configurations

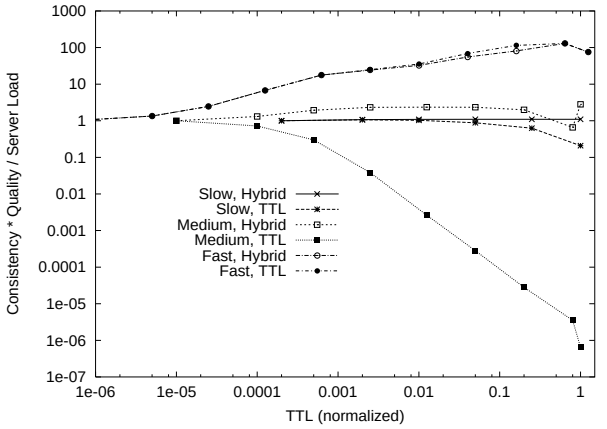

Figure 11: Simple Cost-Benefit Analysis

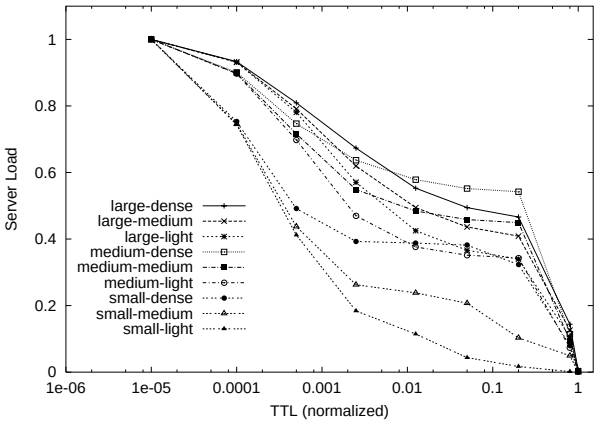

Figure 13: BTC Load Effects for All Configurations

11 we present the output of a simple cost-benefit equation, $Consistency^{\alpha} \times Quality^{\beta} \times ServerLoad^{\gamma}$, where $\alpha = \beta = 1.0$ and $\gamma = -1.0$; This value is one indicator of the consistency and freshness-quality payoff per unit of incurred server load.[14] This graph illustrates a number of things: the approximate equivalence of BTC and TTL for very small TTL values, the relative strength of BTC when cache TTLs exceed object lifetimes, and the ability of BTC to make better marginal use of its communication with the server under most conditions. One surprise in this graph is that TTL slightly outperforms Hybrid for a range of large TTL values under the fast workload; this is largely an artifact of the minuscule server load values at these TTLs (see Figure 8), where the Hybrid algorithm requires roughly 40% more cache misses.

Finally, to illustrate how the size and saturation of the ODG influence the load placed on the server, Figures 12 and 13 present the server load values under a medium-rate request stream for nine ODG configurations (large, medium, and small size; dense, medium, and light saturation).

For the TTL algorithm (Figure 12), ODG saturation naturally has no effect upon the cache hit rate (and thus, no effect upon server load); only the number of resources matters, that having the intuitive effect of making it harder for the cache to shed all of the server's load as the number of resources increases.

The Hybrid algorithm (Figure 13) is virtually identical to TTL for the first order of magnitude of TTL values, but from there differentiates according to ODG saturation with more-saturated configurations always provoking higher server loads (since any single update under such configurations is likely to cause the invalidation of a larger number of cached resources) within each size class.

5 Conclusions

We have described a novel mechanism, Basis Token Consistency (BTC), which uses a logical vector clock mechanism and response annotation to provide strong consistency via lazy notification to any participating cache regardless of the presence and participation of intermediaries; we also briefly discussed several extensions to this protocol which offer server applications and caches greater flexibility in satisfying system goals. We then presented results from a simple simulation of a modern Content Management System (CMS) driving a set of traditional (TTL) and BTC caches and compared their behaviors under a range of parameters, illustrating some of the tradeoffs and effects of TTL values for traditional and BTC caches in terms of their ability to shed server load and the freshness, quality, and consistency of the response streams delivered by

[14] These parameters can be altered to prioritize among the three performance metrics; many other equations are also perfectly reasonable, depending upon how one wishes to measure marginal value.

each.

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

Acknowledgements

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

SDG.

References

[1] Paul Barford, Azer Bestavros, Adam Bradley, and Mark Crovella. Changes in web client access patterns : Characteristics and caching implications. *World Wide Web*, 2:15–28, 1999.

[2] Paul Barford and Mark Crovella. Generating representative web workloads for network and server performance evaluation. In *ACM SIGMETRICS*, 1998.

[3] Adam D. Bradley and Azer Bestavros. Basis token consistency: A practical mechanism for strong web cache consistency. Technical Report BUCS-TR-2001-024, Boston University Computer Science, 2001.

[4] Adam D. Bradley and Azer Bestavros. Basis token consistency: Supporting strong web cache consistency. In *Global Internet Worshop*, Taipei, November 2002. (to appear).

[5] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. On the implications of Zipf's law for web caching. In *3rd International WWW Caching Workshop*, Manchester, England, June 1998.

[6] Ramon Caceres, Fred Douglis, Anja Feldman, Gideon Glass, and Micahel Rabinovich. Web proxy caching: The devil is in the details. In *ACM SIGMETRICS Performance Evaluation Review*, December 1998.

[7] Pei Cao and Chengjie Lui. Maintaining strong cache consistency in the world-wide web. In *ICDCS*, 1997.

[8] Jim Challenger, Arun Iyengar, Karen Witting, Cameron Ferstat, and Paul Reed. A publishing system for efficiently creating dynamic web content. In *INFOCOM (2)*, pages 844–853, 2000.

[9] Jim Challenger, Aryun Iyengar, and Paul Dantzig. A scalable system for consistently caching dynamic web data. In *Proceedings of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies*, New York, New York, 1999.

[10] Fred Douglis, Anja Feldman, Balachander Krishnamurthy, and Jeffrey Mogul. Rate of change and other metrics: a live study of the world wide web. Technical Report 97.24.2, AT&T Labs-Research, December 1997.

[11] C. Fidge. Logical time in distributed computing systems. *Computer*, 24(8):28–33, August 1991.

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

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

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

[15] Balachander Krishnamurthy and Craig Wills. Piggyback server invalidation for proxy cache coherency. In *Proceedings of the WWW-7 Conference*, pages 185–194, Brisbane, Australia, April 1998.

[16] D. Li, P. Cao, and M. Dahlin. WCIP: Web cache invalidation protocol, March 2001. Internet Draft (work in progress) draft-danli-wrec-wcip-01.txt.

[17] Dan Li and Pei Cao. WCIP: Web cache invalidation protocol. In *5th International Web Caching and Content Delivery Workshop*, Lisbon, Portugal, May 2000.

[18] F. Mattern. Virtual time and global states of distributed systems. In *Proc. Parallel and Distributed Algorithms Conf.*, pages 215–226, 1988.

[19] J. C. Mogul, B. Krishnamurthy, F. Douglis, A. Feldmann, Y. Goland, A. an Hoff, and D. Hellerstein. Delta encoding in HTTP, January 2002. RFC3229.

[20] Jeffrey C. Mogul. Clarifying the fundamentals of HTTP. In *WWW-2002*, Honolulu, HI, May 2002.

[21] Mike Reddy and Graham 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.

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

[23] Jian Yin, Lorenzo Alvisi, Mike Dahlin, and Arun Iyengar. Engineering server-driven consistency for large scale dynamic web services. In *WWW10*, Hong Kong, May 2001.

[24] Haifeng Yu and Amin Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of Operating Systems Design and Implementation (OSDI)*, October 2000.

[25] Xiaohui Zhang. Cachability of web objects. Technical Report BUCS-2000-019, Boston University Computer Science, 2000.

[26] Huican Zhu and Tao Yang. Class-based cache management for dynamic web content. In *IEEE INFOCOM*, 2001.

[27] G K Zipf. *Psycho-Biology of Languages*. Houghton-Mifflin, 1935.