Basis Token Consistency

A Practical Mechanism for Strong Web Cache Consistency*

Adam D. Bradley and Azer Bestavros
Computer Science Department
Boston University
Boston, MA 02215
{artdodge,best}@cs.bu.edu

BUCS-TR-2001-024

October 30, 2001

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 becoming increasingly significant and drawing the attention of the standards community [LCD01]. Toward this end, we present definitions of consistency and coherence for web-like environments, that is, distributed client-server information systems where the semantics of interactions with resources are more general than the read/write operations found in memory hierarchies and distributed file systems. We then present a brief review of proposed mechanisms which strengthen the consistency of caches in the web, focusing upon their conceptual contributions and their weaknesses in real-world practice. These insights motivate a new mechanism, which we call "Basis Token Consistency" or BTC; when implemented at the server, this mechanism allows any client (independent of the presence and conformity of any intermediaries) to maintain a self-consistent view of the server's state. This is accomplished by annotating responses with additional per-resource application information which allows client caches to recognize the obsolescence of currently cached entities and identify responses from other caches which are already stale in light of what has already been seen. The mechanism requires no deviation from the existing client-server communication model, and does not require servers to maintain any additional per-client state. We discuss how our mechanism could be integrated into a fragment-assembling Content Management System (CMS), and present a simulation-driven performance comparison between the BTC algorithm and the use of the Time-To-Live (TTL) heuristic.

*This research was supported in part by NSF (awards ANI-9986397 and ANI-0095988).

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 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 as massive mental energies have been expended examining the "problem" of web caching is that the traditional bad guy of the literature, poor eviction and replacement algorithms, is not in fact a fundamental obstacle to "good" use of a caching infrastructure.

The reason for this is twofold: First, mass storage prices are plummeting faster than individual users' network use is increasing (where 20GB seemed like a lot of disk space just a few years ago, today commodity drives with almost twice this capacity are being sold with home PCs). Second, as pointed out by a growing body of literature from the applied research community, many cache replacement/eviction events in even moderately-sized caches are *not* driven by capacity misses [CDF+98, KMK99, RF98, Zha00].

In the modern web, many (if not most) eviction events are driven by one or both of two application goals: First, providing clients with a *consistent* view of the state of the system as it is updated by user-driven events. Second, providing clients with a *recent* view of the state of the system as it is updated by events asynchronous with the user's actions.

There is a healthy body of literature discussing consistency and coherence in distributed file systems [Cat92, How88, SHN+85, SKK+90, Guy91], and the emergence of the field of mobile computing has further driven research in this area [JEHA97, Ebl98, HL97]. Web caching in general has received a great deal of attention in the literature for some time [ASA+95, KLM97, AFA97, DMF97, Abd98]; however, in practice a number of non-obvious issues have been uncovered [CDF+98, KMK99, KR99, Zha00] which pose problems for the simplistic approach taken by traditional web caches. Even the current version of the web's foundational protocol, HTTP/1.1 [FGM+99], does not incorporate any non-trivial consistency or coherence mechanisms [KR01], although there has been some discussion in the literature about ways of adding this [CL97, KW98, ZY01, LC00][1], and there is a current (as of this writing) internet draft [LCD01] which describes how one mechanism could be implemented.

This paper is an attempt to examine and address these current weaknesses and issues systematically. We begin with a brief synopsis of research into adding consistency and coherence mechanisms to the web's application protocol suite; different approaches are examined in light of both their effects and their deployment requirements. We then present a novel mechanism we call Basis Token Consistency (BTC) which captures the desirable properties of its predecessors without violating the spirit or letter of the HTTP protocol. We then conclude by discussing a model for integrating this mechanism with a modern Content Management System (CMS), and use this model to drive simulations which illustrate the performance of the BTC algorithm under varying conditions.

2 Consistency and Coherence within a Web-like Framework

The web does not behave like a distributed file system. Of particular note are the absence of a well-defined "write" event, the inability to "batch" user-provided updates, the complexity of addressing particular content, and the absence of any protocol-layer persistent state or notion of "transactions". In a distributed file system, the mapping from write events to eventual changes in the "canonical" system state is clearly defined; in the web, non-idempotent requests from users can have arbitrary application-defined semantics with arbitary

[1] Some of the concepts therein are also hinted at in [BC96, ZFJ97]

scopes. For this reason, the definitions of consistency and coherence used in the distributed file system literature are not well suited to systems like the web.

In this section we will define our basic terminology and offer definitions of consistency and coherence which are suitable to distributed information systems with the web's more general update semantics. We will also define some basic properties of collections or networks of web caches which will be of interest throughout the paper.

2.1 Entities, Resources, and Caching Networks

Much of the research literature tends to play fast and loose with web terminology, some of the worst victims being the highly overloaded terms *page* and *document*. In the context of this paper we avoid these and use the technical terms *entity* and *resource*.

An *entity* is a byte stream of finite length, intuitively similar to the idea of a "file." Entities are usually found in responses to HTTP requests, although they may also be sent as part of requests. An entity is the "what" that is held by a cache.

A *resource* is an object[2] which is accessible via an application-layer internet protocol and identified by a URI (Uniform Resource Identifier). In the web, a resource performs actions and outputs entities in response to requests; a resource is the "who" caches interact with to acquire entities.

When orienting yourself within a graph of web servers, proxies, and clients, "upstream" is defined as "in the direction of the server" and "downstream" is defined as "in the direction of the client". When referring to proxies with attached caches, we use the "proxy-cache" moniker. Where we talk about "caches," we include both proxy-caches and caches embedded in edge clients. We use the term "clients" to refer to anything which can talk to a server, including proxies and proxy-caches. We use the term servers exclusively for origin servers, not proxies acting in their proxy-server capacity.

2.2 Consistency

For our purposes, cache *consistency* refers to a property of the entities served by a single logical cache to a single logical client, such that no entity will be output which reflects an older state of the server than that reflected by previously output entities.[3] Another way of stating this is that a consistent cache provides to each client a *non-decreasing* view of all data the server uses to construct its output; once you have seen the result of some event having happened, you will never see an entity which suggests that event has not yet happened.

The key insight to differentiating the traditional definitions of consistency and our definition is that the entities carried in web transactions are not themselves the objects with respect to which consistency must be maintained. Rather, entities are derived from pieces of server state; therefore, consistency is defined with respect to the units of this data (we call each unit an "origin datum") which are hidden inside the origin server and connected to resources and entities by some application logic (such as a collection of scripts) as illustrated in Figure 1.

Strictly interpreted, our definition can be thought of as a formulation of "view consistency" [Goe96], in that different versions of entities can be served to different clients, so long as the cache guarantees each client an internally self-consistent "view" of the server's state. Almost all web cache consistency approaches today do not concern themselves with differentiated, self-consistent views; their goal is for the one output stream of the cache to be recent and internally self-consistent. This desire for recency would contradict

[2] Including all of that term's inherent ambiguity.
[3] This is, roughly, the definition used in [CIW+00] and its related papers.

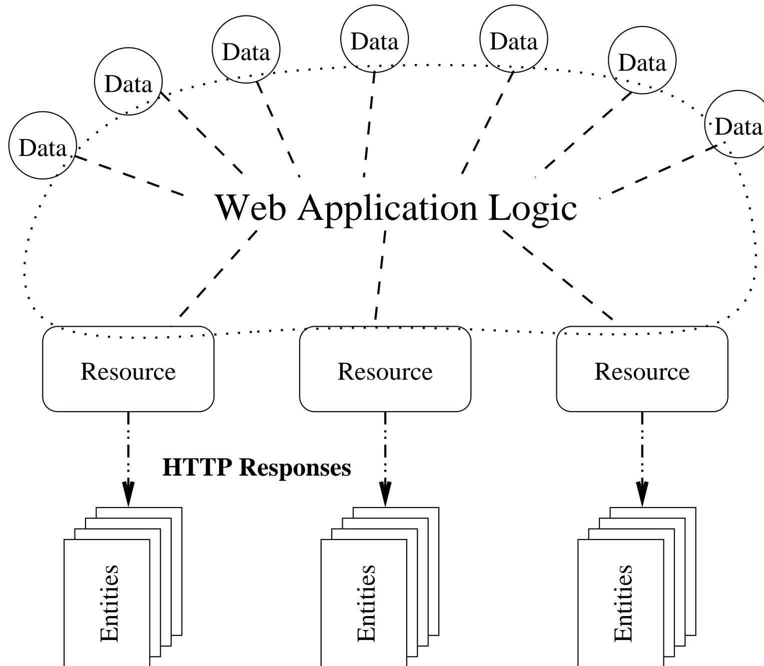

Figure 1: Data, Application Logic, Resources, and Entities

retransmitting older versions to some clients; further, guaranteeing a single internally consistent response stream is sufficient (but not necessary) to the goal of per-client consistency. The mechanism and algorithms for strong consistency we present in section 4 are framed with this same special case in mind, although they can be easily generalized to provide differentiated self-consistent views to individual clients if so desired.

The most common approach taken to providing consistency (when it is dealt with at all) is for a server to disable caching of all entities it serves ("cache-busting"). Granted, this prevents network caches from reaching inconsistent states; however, all the potential benefits of caching are likewise discarded, leaving only the limited and specialized performance benefits of application-layer proxies [FCD+99].

While such overkill is appealing in that it guarantees our goals, it should be clear that as a solution it is *sufficient* but not *necessary* to address the problem at hand. To achieve our goal, all that is really required is a way to communicate the passing of events which obsolete entities currently held by the cache, and the ability to identify all entities which are rendered obsolete by those events.

Unlike some other definitions of consistency used in the web literature, ours is completely independent of the *recency* of cached content; a cache can hold a set of entities that are all months old, but all reflect the server's state at a single point in time, and the cache is thus internally consistent. The need for views to not only be internally consistent but to reflect *recent* server state is thus addressed by cache *coherence*.

2.3 Coherence

In the web, a *cache coherence protocol* is a means for taking updates to entities produced by either user actions or server-side events, and propagating them through the caching network such that all clients interested in entities effected by those updates will eventually see their results; the meaning of the word "eventually" is determined by the choice of coherence protocols.

In the web's current state all origin data reside "behind" origin servers. As such, all updates must reach the root of the distribution network before they affect any entities anywhere in the system; thus, the web

requires a "write-through" approach to updates. In such a system, a coherence protocol's job is simply to limit or inhibit a cache's ability to retain and reuse entities derived from versions of server data which have been obsoleted.

Contrast this with traditional caching environments, where it is possible for update events to definitively change the contents of a cache before the "authority" (main memory, upstream server, origin server) is made aware that an update has happened. These updates can then be queued up, batched, and delivered upstream as determined by a "write-back" coherence protocol. While there are many who believe the web may be moving toward such a world [Not01, BH01], this is far from becoming a deployed reality in non-proprietary systems, and as such is not discussed further in the context of this paper.

There are essentially two coherence models in the current web. The first is "immediate coherence" in which a cache should never return an entity that is stale. This is the ultimate in semantic transparency, and as a side-effect is guaranteed to provide a consistent view of the server's state; however, in practice it demands communications between caches and the origin server which are either frequent or out-of-band[4].

The second is "bounded staleness"; this is engendered in the use of expiry mechanisms in the current HTTP protocol to limit how stale a cached entity can become, guaranteeing that no entity will ever be more than a known timespan out-of-date.

If this latter mechanism is added to a correct consistency mechanism, we would expect the average amount of staleness that can accumulate to be much lower than that suggested by the staleness bound for entities derived from widely-used origin data. Consider that every time the cache "misses" and fetches an entity dependent upon such data, it is an opportunity for a consistency-based invalidation event prior to the pre-set expiration time. In contrast, consider what would be required for the expiration time to be the actual bound upon staleness: the entire common-data-sharing working set would need be brought into the cache while the base data are in a single state (seeing no updates), after which the user remains in a closed and idempotent browsing pattern within that working set. We discuss in section 5 how this is possible in practice under certain conditions.

2.4 Structure of Caching Networks

Note that the topology of a caching network with respect to a given server and its clients can play a significant role in the efficiency and correctness of both consistency and coherence protocols. Of particular concern are two issues: upstream path divergence and downstream fanout.

Upstream path divergence is the property of the cache network upstream of a node such that requests directed to a single logical server could take different logical path through that network. This is problematic in that it becomes possible for the response stream seen at the "split point" to be internally inconsistent even if all caches along each of its 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 same server state at the same time. 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.

[4] That is, out-of-band with respect to the web's stateless client-server messaging model.

The fanout property pertains to the number of clients who make requests directly to any given node. This number becomes a scalability issue where upstream nodes need to maintain some per-client state in order to support consistency and coherence mechanisms in downstream caches, as many of the previously proposed mechanisms discussed here do; given a large number of clients, we need to either minimize per-client state or maintain a small fanout using layers of intermediate proxies (as is suggested in [LCD01]).

3 Related Work

With this framework of goals and priorities in place, we now examine several mechanisms proposed in the literature to provide a consistent and/or more coherent web. We have translated their terminologies into ours as needed.

3.1 Traditional Consistency Approaches

The HTTP protocol as specified contains no mechanisms for explicit coherence-related actions, or for enforcing any consistency across resources. Thus, where consistency and coherence are required, three techniques are generally applied:

1. **Ignore the problem** - Many sites in the current present-day web are designed with the tacit assumption that the only caching that will take place is that done by user-agents themselves, and that it is easily overridable by clicking the "Reload" button; this will not necessarily work in the presence of standards-compliant proxy-caches.

2. **Disable all caching** (a.k.a. cache busting) - While this causes the system to behave correctly, the removal of the beneficial effects of caching (including the browser's integrated cache) may aggravate performance issues, particularly for users who make regular use of their "Back" button as a navigational tool. Which leads us to:

3. **Forbid use of the "Back" button** - This approach allows servers to work around any intermediary caches by using a new URI (resource name) for every step of a user's interaction, rendering caching (which is indexed on URI) moot. This is undesirable from a user interface standpoint as use of the Back button is common practice [Tau96, TG].

Further, the only available coherence model in the protocol is per-entity lifespans or expiration times set when an entity is transmitted by the origin server; thus, the amount of time which an entity can spend in the "stale" state has an upper bound set at the time of its creation, a time when an accurate prediction of the entity's actual longevity may be difficult to make [DFKM97, RF98].

These mechanisms do have the desirable properties that they work independently of the topology of caches between server and user; however, what they provide is at best a crude simulation of true consistency.

3.2 Proposed Consistency and Coherence Mechanisms

A body of work has emerged proposing replacement mechanisms which more directly address consistency and coherence problems. Some distinguish between the two issues; others treat them as a single problem (either explicitly or implicitly).

The first three examples (Adaptive TTL, Polling-every-time, and Active invalidation) are presented in a paper by Cao and Liu [CL97]. They define web consistency as "cached copies should be updated when

the originals change"[5]; under our terminology, this is *immediate coherence* (that is, changes are propagated immediately) which, as mentioned earlier, provides strong consistency as a side effect when perfectly realized.

3.2.1 Adaptive TTL

The premise of the Adaptive Time-To-Live (TTL) approach (derived from [Cat92]) is that as resources see their state updated (or as caches observe changes in the entities provided by a resource), they alter the cache lifespan of the entities they serve to reflect a better estimation of "true" lifespans [RF98]. This is of course only a heuristic, and as such it provides only weak consistency with no guarantees; neither does it have any explicit coherence strategies as such beyond per-entity expiry. The inability of the system to "adapt" or "revoke" over-estimates of short lifespans also prevents it from making any guarantees.

In its favor, the adaptive TTL approach is completely compatible with the current web infrastructure, and can be implemented in servers as well as in caches without any explicit cooperation between nodes.

3.2.2 Polling Every Time

The second technique discussed in [CL97] is "polling-every-time" (PET), in which cached entities are pessimistically checked against an upstream node using HTTP's validation mechanism[6] every time they are accessed. It is an approximation of *immediate coherence* mechanism, and thus approximates strong consistency as well.

We say it is an approximation because a cache implementing this strategy can only ever be "as consistent as its upstream provider"; it will never provide an entity that is view-inconsistent unless the upstream node would provide that same inconsistent entity. As such, this technique is particularly susceptible to problems when the upstream topology diverges and includes any non-PET nodes; it has no way of detecting internal inconsistencies within a response stream.

In addition to its reliance upon the consistency and coherence of upstream nodes, this approach also suffers from an inability to propagate its own (hopefully) self-consistent state through downstream caches that do not implement PET, except by deliberately busting downstream caches [Bra01].

3.2.3 Active Invalidation

The active invalidation approach is the third family of consistency techniques discussed in [CL97][7]. To implement it, a server keeps track of all clients who have received entities from it so that when any entity is altered invalidation messages can be immediately sent to all such clients. This approach is used in the Andrew File System [How88].

Once again, this is a hybrid consistency/coherence mechanism; it aims for strong immediate coherence, but suffers from race conditions between resource updates on the server and the receipt of invalidation messages. Since the consistency of this scheme can only be as tightly synchronized as its coherence (e.g. it relies upon coherence events arriving simultaneously to guarantee internal consistency), its consistency is therefore also "weak" as it is possible for the cache to provide inconsistent results while invalidations are in flight, even if the upstream node is always self-consistent.

Topology is a significant issue when it comes to building an active invalidation caching network in that each cache relies upon its entire upstream path to propagate invalidation messages to it. Asymmetric

[5] Similar terminology also appears in [LCD01], of which Cao is also a co-author; that proposal is not discussed in this paper, as its underlying concepts are mostly present in the papers already discussed.

[6] HTTP validation incurs the delay of a network request-response transaction, but saves the cost of re-transmitting an identical entity from the same resource.

[7] There called simply "Invalidation".

reachability (firewalls and NAT boxes) is an obstacle; degree of fanout of the network directly affects the resource demands for tracking which nodes require invalidations, although several optimizations can be applied which may keep this cost manageable.

3.2.4 Class-Based Invalidation

Proposed by Zhu and Yang in [ZY01], this mechanism integrates several key ideas we will build upon in our own proposed technique. Three key concepts we highlight here are *aggregate invalidation messages*, *abstraction of dependence*, and a *lazy invalidation strategy*.

The simultaneous *aggregate invalidation* of all entities from a set of resources greatly improves the efficiency of consistency mechanisms, and eliminates (potentially, anyway) race conditions between the invalidation of simultaneously outdated entities. The class-based approach allows the server to specify sets resources (by URI prefix) and requests parameters which identify a "class" which can be invalidated as a whole.

By *abstraction of dependence*, we mean the ability to identify sets of resources as depending upon a single origin datum, and then invalidating all entities from such resources using a single group identifier which is independent of the URI namespace. This is essential for performing aggregate invalidations when the target resources cannot be accurately grouped by URI prefixes or similar constructs.

Finally, the *lazy invalidation strategy* moves us back toward the client-server communication model of HTTP, where downstream nodes only receive messages from upstream nodes in response to requests. For each caching node in the network that may hold stale entities, the mechanism embeds appropriate necessary entity, class, and abstraction-based eviction messages in the next response sent to that node (independent of what resource the request was for).

Class-based invalidation interacts poorly with non-supporting intermediary caches; because the invalidation message stream is "overlayed" on what is an independent data stream (responses), intermediaries are unable to separate the streams and de-multiplex them to all of their own clients that might need to receive them.

There are also the inherent scalability problems of the per-client state to queue up the required invalidation messages, although its storage requirement scales per-user and per-update rather than per-user and per-resource as in the invalidation case above.

3.2.5 Piggyback Invalidation

Piggyback server invalidation (PSI) is proposed by Krishnamurthy and Wills in [KW98] also presents a *lazy invalidation strategy* for web caches. Their approach integrates several ideas which make it preferable to class-based cache management.

To implement PSI, the server's resources are grouped into "volumes." The policy for doing so is not important to the correctness of the mechanism, although it does affect efficiency. Every response is annotated with the volume to which it belongs, and with a version number which is incremented whenever the contents of the volume are changed. Clients retain these volume identifiers and version numbers, and send them back as part of validation requests.

When the server observes a request for a resource which bears an out-of-date volume version number, it compiles a list of the resources in that volume that have changed since the client's version number was current and includes ("piggybacks") that list in the response, telling the client's cache to invalidate any entities it has acquired from any of those resources.

The main reason for our preference of this approach over the Active Invalidation and Class-Based techniques is that the server is no longer required to maintain any persistent per-client state, making the approach

inherently scalable; the only state that must be maintained is the per-volume invalidation history, earmarked with volume version numbers, and that is completely independent of number of clients.

A significant benefit of the PSI approach is the ability to identify inconsistent (stale) results by ensuring that the volume version number attached to every response is no less than the current (highest) version number seen for that volume. Thus, an end cache can still produce consistent output even if its upstream cache is not conforming (or if the upstream topology diverges).

4 Basis Token Consistency

Taking key ideas from the above literature, we have devised a web cache consistency protocol we call "Basis Token Consistency" (BTC) with the following properties: (1) Strong point-to-point consistency does not rely on intermediary cooperation. (2) No per-client state is required at the server or intermediaries. (3) Invalidations are aggregated, and aggregations are independent of HTTP namespaces. (4) Invalidation is driven by the application, not heuristics. (5) The necessary data is attached only to salient entities, and is thus lazily delivered with no "overlay" problem.

4.1 Conceptual Overview of BTC

In our approach, entities in responses are annotated with enough information to allow any cache interested in maintaining consistency to detect if other entities it currently holds are obsoleted by that entity, and to recognize entities it may receive in the future which are already stale relative to entities previously received.

To accomplish this, each entity bears the `Cache-Consistent` header with a set of *basis tokens*, each with a generation number. Each basis token represents some dynamic source of information in the underlying application; each such source is called an origin datum. All entities which depend upon a particular origin datum will include its basis token, and whenever a datum is changed, its generation number is incremented, and all entities produced using that datum in the future will reflect the new generation number.

Caches implementing BTC keep an additional index of entities keyed on their basis tokens. Whenever a new entity arrives, each token's generation number is checked against the cache's "current" generation number for the same token. If they match, no further action is taken. If the new 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 new generation number is less, then the entity is stale and inconsistent; the request should be repeated using an end-to-end validation or reload.

4.2 Implementing BTC

The BTC mechanism uses a single new HTTP entity header, `Cache-Consistent`; the augmented-BNF[8] grammar for this header is given in Figure 2. `cctoken` is the basis token identifier, expressed as an opaque string; `ccgeneration` is the generation number, expressed as a hexadecimal integer.

To allow common data to be used in the backends of multiple servers and prevent collisions across disjoint servers, tokens can be scoped to particular hostnames by providing a *cctokenscope*. The scope defaults to the fully-qualified domain name of the host sending the token. When looking for matching tokens, the scope string is always used, whether it was explicit or implicit. Thus, if the server cs-people.bu.edu sends this header with a response:

[8] Augmented-BNF is the *ad hoc* standard form for the specification of text protocol grammars in RFCs. For a summary, refer to [FGM+99] §2.1.

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

Figure 2: The Proposed `Cache-Consistent` HTTP Entity Header

`Cache-Consistent: db1row;19, db2row@bu.edu;7`

and the server www.bu.edu sends the header with an entity:

`Cache-Consistent: db1row;2c, db2row;9`

then none of the tokens match, because the implicit scopes for `db1row` are `@cs-people.bu.edu` and `@www.bu.edu`, respectively, and the implicit scope of www.bu.edu's db2row token is `@www.bu.edu`, which mismatches the explicitly scoped `@bu.edu` token given by cs-people.

If, however, www.bu.edu later sends an entity with the header:

`Cache-Consistent: db1row;2c, db2row@bu.edu;9`

Then the previous entity from cs-people.bu.edu is invalidated because of the match of the second token and its higher generation number. (There is still no match between their `db1row` tokens, however.)

To frustrate certain kinds of denial-of-service attacks, an entity may only scope token to its own fully-qualified domain name or a superdomain (suffix) thereof. Tokens violating this rule are discarded by all downstream clients. The idea is that token scopes correspond with (at least nominal) administrative scopes; attempts to scope outside of your own administrative hierarchy (www.cs.unca.edu trying to use `@eng.unca.edu`) will be rejected outright, and scoping a token "above" your administrative control (www.bu.edu trying to use `@edu`) is prone to collisions and should thus be avoided.

4.3 Limitations of BTC

Large Stack Distances: The BTC algorithm's ability to guarantee the self-consistency of an output stream is limited by its ability to retain a history of token generation numbers seen in the past; when the span (stack distance) of accesses to consistency-related resources exceeds the capacity of this history mechanism, inconsistencies introduced upstream go unnoticed. It is hoped that such history mechanisms will have adequate capacity that traditional expiration will kick in before this problem emerges.

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 not excessive in either space or 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.[9] As

[9] It would make intuitive sense for their popularity to follow a Zipf-like distribution [Zip35]; while we lack evidence from any reasonably large sampling of "real" sites, the data presented in figure 11 of [CIW+00] lends some support to this theory.

such, we expect the number of tokens in the index to grow linearly with the number of URIs[10]. Since scoped tokens occupy a flat global namespace, entities can be indexed on their tokens using the same data structures (with the same complexity) as they are indexed on URIs, yielding a constant-factor increase in index storage space and update complexity.

Reliance on consistency of application: Unlike other approaches, BTC simply cannot work without some sort of application support, and depends upon those applications being internally correct and consistent. Consider, for example, the case where a poorly-designed web application attaches two basis tokens, A and B, to an entity. Should a BTC cache receive one copy of the entity where A's generation is 1 and B's generation is 2, and another copy where A's generation is 2 and B's generation is 1, the cache is unable to discern which is the "newer" copy, and its behavior becomes undefined. A sane policy might be to flush all entities depending on any generation of A or B and start from scratch, or to refuse to cache entities from this server after such an inconsistency is detected.[11]

Non-implementing Clients: While BTC in itself guarantees that any cache-server pair wishing to participate can maintain view consistency, this guarantee does not extend end-to-end because (for example) a non-compliant user agent cache can still introduce inconsistencies to the user's experience. While we are not yet able to address this problem when the network diverges before the closest-to-the-client BTC cache, we can in most cases address it by extending HTTP's `Cache-Control` mechanism with a new parameter, `cc-maxage`. This value is given primacy over the `Expires` header and the standard `max-age` and `s-maxage` parameters in BTC caches; thus, an entity can be pre-expired, "busting" all non-conforming caches, with a special provision allowing only BTC caches to retain and reuse the content.

4.4 Extending BTC

We have defined several extensions to BTC; for example, we have algorithms that allow a BTC cache to provide differentiated view consisteny response streams to clients, relax consistency to accept version ranges, and address issues with inconsistent components of a single page assembled from mulitple entities. We have also added an explicit coherence mechanism borrowing ideas from the class-based approach discussed earlier. For lack of space, these extensions are not discussed in this paper.

5 Validation

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 [DFKM97, RF98]), 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.

[10]More precisely, we expect this in steady state; when a server is first visited, there will be a spike as the "popular" tokens are first introduced.

[11]It bears mentioning that assembling web pages mid-network without strong coherence and consistency measures among the assembly nodes makes the existence of such contradictory entities possible; in the presence of a diverging upstream network, a single client may be able to observe the contradiction.

5.1 Basis Token Consistency and Content Management Systems

Our hypothetical Content Management System (or CMS) is inspired by IBM's DUP-based system [IC98, CID99, CIW+00], primarily because its organization and algorithms are representative of best current practices in industry and well-described in the research literature.

The concept of a CMS is quite straightforward. The entities that make up a modern content-driven web site are usually 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.

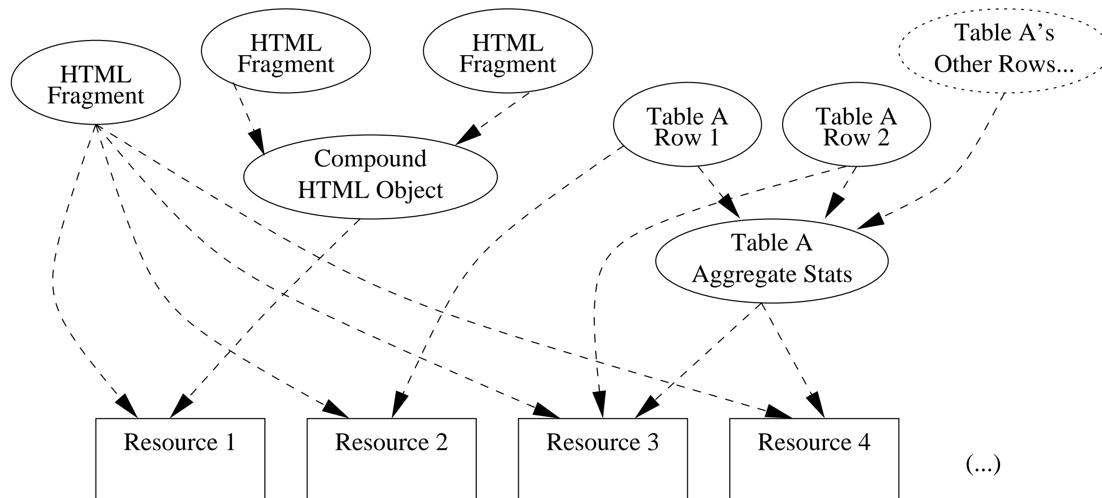

Figure 3: Sample Object Dependence Graph

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 [IC98] 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.[12]

A more attractive solution would be for each resource to only use the nodes of its origin data; while this would also produce correct results, it fails to take advantage of the graph's potential aggregation of clusters of origin data into single "atomic" compound objects.

A preferable solution is for each resource to communicate tokens for only the first "layer" of nodes it depends upon. All updates that cause changes in the entity's content must propagate to a generation number update at this level, thus correctness is preserved.

Of course, this could be taken one step farther; sets of first-layer nodes may always be grouped together when included, suggesting that they could all be represented as a single "virtual" node and thus a single basis token.

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.

5.2 Simulation Design

Lacking any thorough statistical and topological study of the characteristics and morphologies of object dependence graphs found in the wild (the information provided in [CIW+00] is interesting but is not necessarily representative), we implemented a simple CMS server model as a bipartite graph, consisting of datum nodes and resource nodes, parameterized along two axes: size and saturation. For size, we ran experiments with 40, 200, and 400 resource and 200, 1000, and 2000 datum nodes, respectively. The saturation is the percentage of the possible edges that could exist in the bipartite graph that are actually present; we chose values of 12.5%, 25%, and 50%, and build the interconnect by first attaching each resource to a randomly chosen datum and then adding random resource-datum pairs until the desired saturation level is reached. We impose no locality or popularity assumption on this internal structure. In this paper we report the results of the "small and dense" experiments whixh uaw 40 resources, 200 datum nodes, and a 50% saturation.

Each datum is then independently assigned an update process. Our simulator can use a number of different event update models, including exponential, normal, and heavy tailed processes; the results presented

[12] For anecdotal evidence, see figure 10 in [CIW+00].

in this paper reflect all updates being driven by exponential processes whose mean rates were themselves drawn from an exponential distribution. The resulting graph looks something like Figure 4.

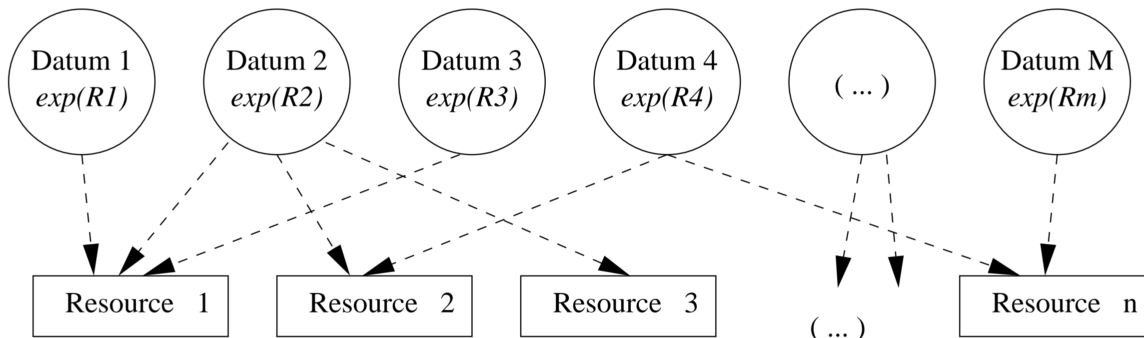

Figure 4: Bipartite ODG and Update Model

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

Resources are next assigned popularities according to a Zipf-like popularity distribution [Zip35, CBC95, BCF^{+}98, BBBC99]; the skew parameter, α, could be varied arbitrarily, but all of our experiments use the value 0.7, reflecting (approximately) the value reported in [BBBC99].

This update event stream is then interleaved with a synthetic request event stream. Requests are targeted at resources randomly in proportion to their popularities. They have a constant inter-arrival time, which is calibrated such that the total number of requests is a multiple of the number of updates (either 1, 20, or 400 in our experiments; these are labeled *slow*, *medium*, and *fast*) and the two substreams have the same duration in time. This ratio allows us to describe sites and caches with a wide range of ratios between request and update rates; for example, a low ratio suggests a fast-changing site with a few slow browsers, while a high ratio suggests intense browsing of a relatively static site.

While this is a very simplistic request model in light of current understanding of web workloads [BC98], 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.[13]

[13] Think of the quality value as the fine-grained counterpart to the discrete "stale/fresh" flag.

5.3 Simulation Results

In this section we present the results of simulations where workloads of varying intensities are run against the small-and-dense server model discussed above. Our choice of parameters to these models and simulations is largely arbitrary; except for the Zipf popularity distribution of resources, we are merely speculating and exploring the parameter space to illuminate the interesting properties of our algorithm. We make no assertion that the particular parameter values reflected here are meaningful or that our results will be representative of common case gains or advantages should BTC be deployed in the wild; they are simply educational.

All graphs present the time-to-live parameter on the X axis, normalized to the length of the simulation in time[14]. The Y axis is normalized to the total number of requests made in a simulation run.

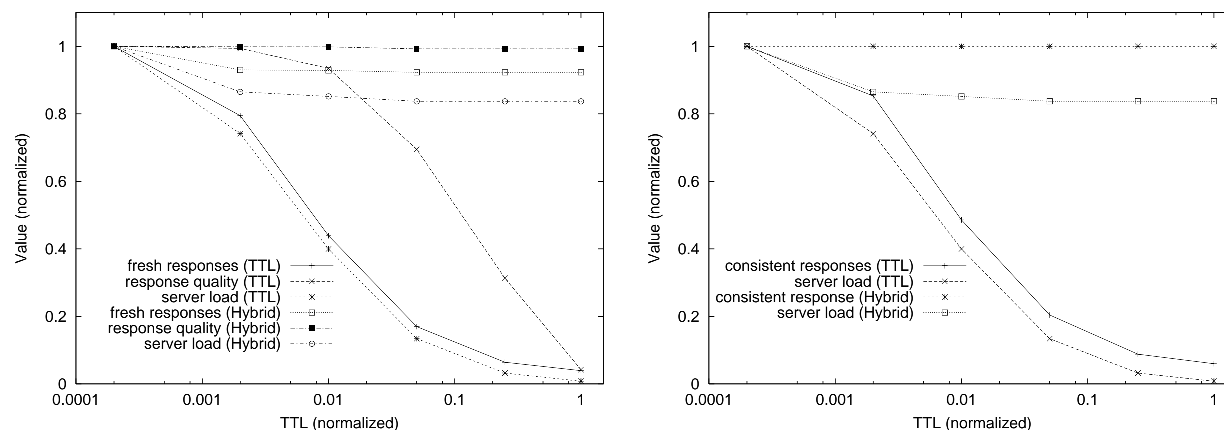

Figure 5: Behavior with Slow Request Stream

Figure 5(left) 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 cache or small-population shared cache. Notice that the TTL algorithm sheds 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 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 fairly large number of origin data, and TTLs tend to expire long before these update events can accumulate to render all of a resource's parts out-of-date.

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.

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 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(right) shows how the cost in server load achieves our design goal of strong view consistency where TTL fails. The "consistent responses" value indicates the number of responses that do no reflect any

[14]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

older versions of origin data than have already been seen by the cache; 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.

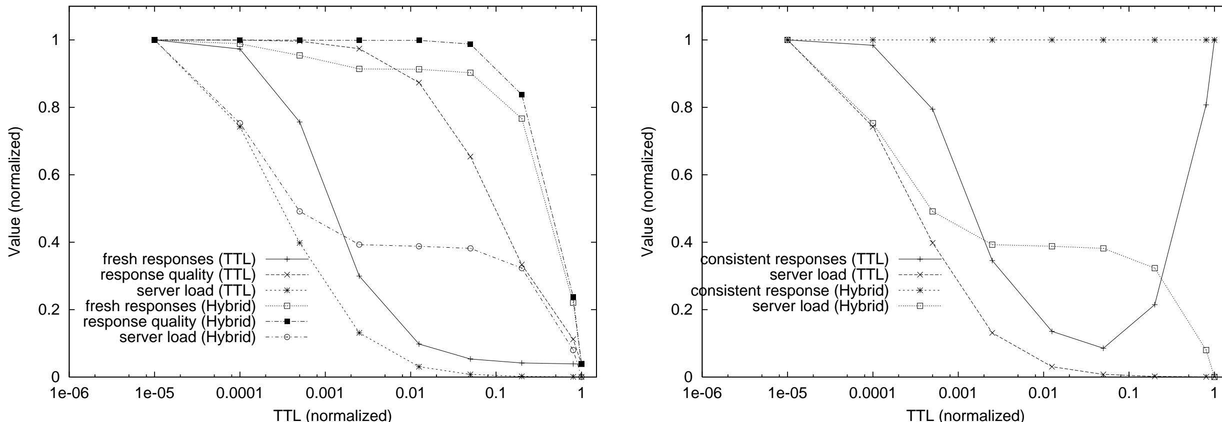

Figure 6: Behavior with Medium-Rate Request Stream

The same setup with a medium-rate request stream shows some very interesting behaviors, as illustrated in figure 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. We also see a multiplicative shift of the quality of TTL responses relative to the stale/fresh metric similar to the one 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 6(right); 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 (lazily delivered) invalidation-provoking data. The same effect causes TTL's consistency value to spike for large TTLs. This is where the interaction 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 rate, or when the Zipf parameter is particularly high; at the same time, high Zipf parameters make it less likely that those rarely-accessed (and thus potential "snapshot breaking") resources will actually be requested.

Finally, we turn our attention to the high request rate run, reflected in Figure 7. 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 valules). 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 8.

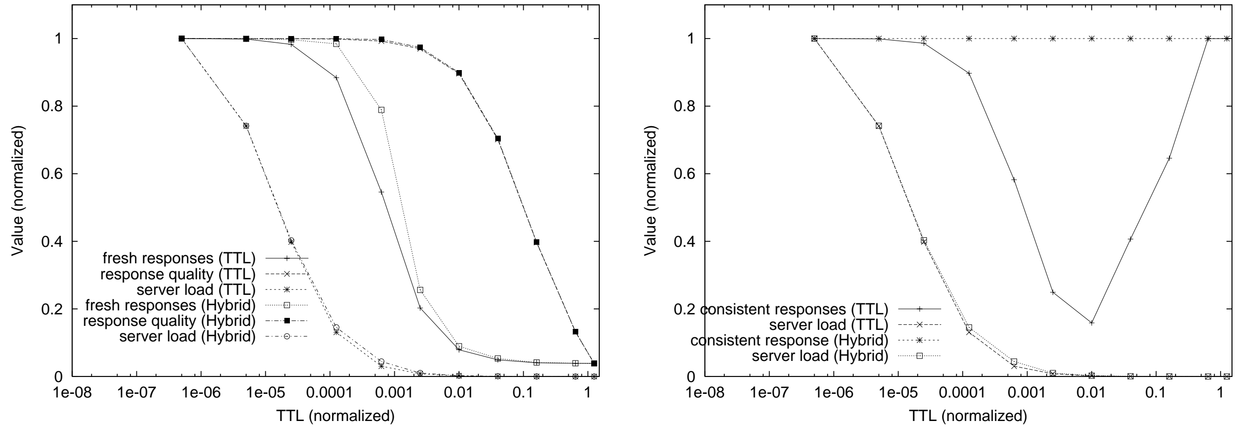

Figure 7: Behavior with Fast Request Stream

6 Conclusion

In this paper, we have provided definitions of consistency and coherence for the web environment which differentiate the problem of maintaining non-contradictory cached views of server state from the problem of propagating updates of server state to clients. We have examined previously proposed mechanisms for adding consistency and coherence to the web, and have identified the issues of *strength* (weak v. strong), *communication models* (immediacy v. lazy notification), *aggregated invalidation*, *abstraction of aggregation* (away from the URI namespace), *scalability* (per-client state), and *interoperability* as they work to the benefit and detriment of each.

We then proposed a novel mechanism, Basis Token Consistency (BTC), which takes advantages of our understanding of those concepts to provide strong consistency via lazy notification to any participating cache regardless of the presence of intermediaries, and discussed how a modern Content Management System (CMS) could integrate BTC in a straightforward manner. Finally, we presented some of our results from simulations of traditional TTL caches and hybrid BTC-TTL caches and illustrated some of the tradeoffs and effects of each in terms of ability to shed server load and the "correctness" of the response stream delivered by each over a wide variety of system parameterizations, focusing particularly upon the effects of varying the request rate with respect to the average backend update rate.

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 reasonable to deploy in the present-day web infrastructure.

Acknowledgements

The authors wish to thank Assaf Kfoury for helpful comments on drafts of this paper.

SDG.

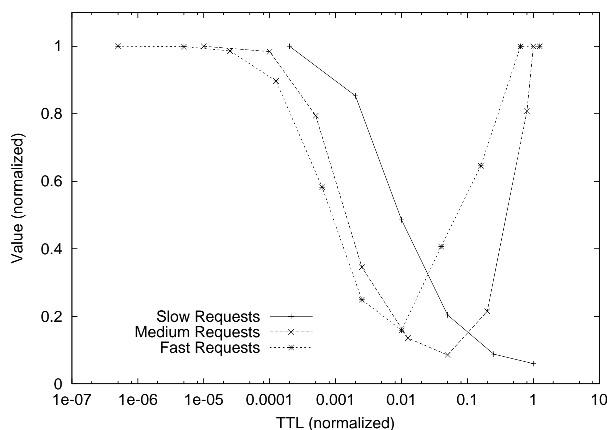

Figure 8: TTL Consistency under Various Request Rates

References

[Abd98] Ghaleb Abdulla, *Analysis and modeling of world wide web traffic*, Ph.D. thesis, Blacksburg, VA, 1998.

[AFA97] G. Abdulla, E. Fox, and M. Abrams, *Shared user behavior in the world wide web*, Proceedings of WebNet97 (Toronto, Canada), 1997.

[ASA+95] Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, Stephen Williams, and Edward A. Fox, *Caching proxies: Limitations and potentials*, Proceedings of 1995 World Wide Web Conference, 1995.

[BBBC99] Paul Barford, Azer Bestavros, Adam Bradley, and Mark Crovella, *Changes in web client access patterns : Characteristics and caching implications*, World Wide Web **2** (1999), 15–28.

[BC96] Azer Bestavros and Carlos Cunha, *Server-initiated document dissemination for the WWW*, IEEE Data Engineering Bulletin **19** (1996), no. 3, 3–11.

[BC98] Paul Barford and Mark Crovella, *Generating representative web workloads for network and server performance evaluation*, ACM SIGMETRICS, 1998.

[BCF+98] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker, *On the implications of zipf's law for web caching*, 3rd International WWW Caching Workshop (Manchester, England), June 1998.

[BH01] Andre Beck and Markus Hofmann, *Enabling the internet to deliver content-oriented services*, Sixth International Workshop on Web Caching and Content Distribution (Boston), June 2001.

[Bra01] Adam D. Bradley, *Comprehensive client-side traces using caching domain isolation*, Tech. Report Work In Progress, Boston University Computer Science, 2001.

[Cat92] V. Cate, *Alex - a global file system*, Proceedings of the 1992 USENIX File System Workshop, May 1992, pp. 1–12.

[CBC95] Carlos R. Cunha, Azer Bestavros, and Mark E. Crovella, *Characteristics of WWW client-based traces*, Tech. Report BU-CS-95-010, Boston University Computer Science, July 18 1995.

[CDF+98] Ramon Caceres, Fred Douglis, Anja Feldman, Gideon Glass, and Micahel Rabinovich, *Web proxy caching: The devil is in the details*, ACM SIGMETRICS Performance Evaluation Review, December 1998.

[CID99] Jim Challenger, Aryun Iyengar, and Paul Dantzig, *A scalable system for consistently caching dynamic web data*, Proceedings of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies (New York, New York), 1999.

[CIW+00] Jim Challenger, Arun Iyengar, Karen Witting, Cameron Ferstat, and Paul Reed, *A publishing system for efficiently creating dynamic web content*, INFOCOM (2), 2000, pp. 844–853.

[CL97] Pei Cao and Chengjie Lui, *Maintaining strong cache consistency in the world-wide web*, ICDCS, 1997.

[DFKM97] Fred Douglis, Anja Feldman, Balachander Krishnamurthy, and Jeffrey Mogul, *Rate of change and other metrics: a live study of the world wide web*, Tech. Report 97.24.2, AT&T Labs-Research, December 1997.

[DMF97] B. M. Duska, D. Marwood, and M. J. Feeley, *The measured access characteristics of world-wide-web client proxy caches*, Proceedings of the 1997 USENIX Symposium on Internet Technologies and Systems (USITS) (Monterey, CA), 1997.

[Ebl98] M. R. Ebling, *Translucent cache management for mobile computing*, Ph.D. thesis, Carnegie Mellon University, March 1998.

[FCD+99] Anja Feldmann, Ramon Caceres, Fred Douglis, Gideon Glass, and Michael Rabinovich, *Performance of web proxy caching in heterogeneous bandwidth environments*, IEEE INFOCOM (1999), 107–116.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, *Hypertext transfer protocol – HTTP/1.1*, 1999, RFC2616.

[Goe96] Ashvin Goel, *View consistency for optimistic replication*, Master's thesis, University of California, Los Angeles, Febrruary 1996, Available as UCLA Technical Report CSD-960011.

[Guy91] Richard G. Guy, *Ficus: A very large scale reliable distributed file system*, Tech. Report CSD-910018, UCLA, Los Angeles, CA (USA), 1991.

[HL97] Q. L. Hu and D. L. Lee, *Adaptive cache invalidation methods in mobile environments*, Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing, August 1997.

[How88] John H. Howard, *An overview of the andrew file system*, Proceedings of the Winter USENIX Conference (Dallas), February, 1988, pp. 23–36.

[IC98] Arun Iyengar and Jim Challenger, *Data update propogation: A method for determining how changes to underlying data affect cached objects on the web*, Tech. Report RC 21093(94368), IBM T. J. Watson Research Center, 1998.

[JEHA97] J. Jing, A. K. Elmargarmid, S. Helal, and R. Alonso, *Bit-sequences: An adaptive cache invalidation method in mobile client/server environments*, ACM/Baltzer Mobile Networks andd Applications, vol. 2(2), 1997.

[KLM97] Thomas M. Kroeger, Darrell D. E. Long, and Jeffrey C. Mogul, *Exploring the bounds of web latency reduction from caching and prefetching*, Proceedings of the Symposium on Internet Technologies and Systems, December 1997.

[KMK99] Balachander Krishnamurthy, Jeffrey C. Mogul, and David M. Kristol, *Key differences between HTTP/1.0 and HTTP/1.1*, Proceedings of the WWW-8 Conference (Toronto), May 1999.

[KR99] Balachander Krishnamurthy and Jennifer Rexford, *En passant: Predicting HTTP/1.1 traffic*, Proceedings of Global Internet Symposium, December 1999.

[KR01] ———, *Web protocols and practice*, Addison-Wesley, 2001.

[KW98] Balachander Krishnamurthy and Craig Wills, *Piggyback server invalidation for proxy cache coherency*, Proceedings of the WWW-7 Conference (Brisbane, Australia), April 1998, pp. 185–194.

[LC00] Dan Li and Pei Cao, *WCIP: Web cache invalidation protocol*, 5th International Web Caching and Content Delivery Workshop (Lisbon, Portugal), May 2000.

[LCD01] D. Li, P. Cao, and M. Dahlin, *Wcip: Web cache invalidation protocol*, March 2001, Internet Draft.

[Not01] Mark Nottingham, *Optimising web services with intermediaries*, Sixth International Workshop on Web Caching and Content Distribution (Boston), June 2001.

[RF98] Mike Reddy and Graham P. Fletcher, *Intelligent web caching using document life histories: A comparison with existing cache management techniques*, 3rd International WWW Caching Workshop (Manchester, England), June 1998.

[SHN+85] M. Satyanarayanan, J. H. Howard, D. N. Nichols, R. N. Sidebotham, A. Z. Spector, and M. J. West, *The itc distributed file system: Principles and design*, Proc. 10th ACM Symposium on Operating Systems Principles, December 1985, pp. 35–50.

[SKK+90] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere, *Coda: (a) highly available file system for a distributed workstation environment*, IEEE Transactions on Computers (Washington, DC), vol. 39, IEEE Computer Society, 1990, pp. 447–459.

[Tau96] L. Tauscher, *Evaluating history mechanisms: An empirical study of reuse patterns in world wide web navigation*, Master's thesis, Alberta, Canada, 1996.

[TG] L. Tauscher and S. Greenberg, *How people revisit web pages: Empirical findings and implicaations for the design of history systems*, International Journal of Human Computer Studies **47**, no. 1.

[ZFJ97] Lixia Zhang, Sally Floyd, and Van Jacobson, *Adaptive web caching*, NLANR Web Cache Workshop (Boulder, CO), 1997.

[Zha00] Xiaohui Zhang, *Cachability of web objects*, Tech. Report BUCS-2000-019, Boston University Computer Science, 2000.

[Zip35] G K Zipf, *Psycho-biology of languages*, Houghton-Mifflin, 1935.

[ZY01] Huican Zhu and Tao Yang, *Class-based cache management for dynamic web content*, IEEE INFOCOM, 2001.