This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2479222, IEEE Transactions on Parallel and Distributed Systems

Measuring and Evaluating Live Content Consistency in a Large-Scale CDN

Guoxin Liu, Haiying Shen*, *Senior Member IEEE*, Harrison Chandler, Jin Li, Fellow, IEEE

Abstract—Content Delivery Networks (CDNs) play a central role in today's Internet infrastructure and have seen a sharp increase in scale. More and more internet sites are armed with live contents, such as live sports game statistics, e-commerce, and online auctions, and they rely on CDNs to deliver such contents freshly at scale. However, the problem of maintaining consistency for live (dynamic) contents while achieving high scalability is non-trivial in CDNs. The large number of widely scattered replicas guarantees the QoS of end-users while substantially increasing the complexity of consistency maintenance under frequent updates. Current consistency maintenance infrastructures and methods cannot simultaneously satisfy both scalability and consistency. In this paper, we first analyze our crawled trace data of cached sports game content on thousands of content servers of a major CDN. We analyze the content consistency from different perspectives, from which we break down the reasons for inconsistency among content servers. We verify that the CDN uses unicast instead of multicast trees as the update infrastructure, which may not scale effectively. Then, we further evaluate the performance in consistency, scalability and overhead for different infrastructures with different update methods. We itemize the advantages and disadvantages of different methods and infrastructures in different scenarios through the evaluation. Based on this evaluation, we propose our hybrid and self-adaptive update method to reduce network load and improve scalability under the conditions recorded in the trace and prove its effectiveness through trace-driven experiments. We aim to give guidance for appropriate selections of consistency maintenance infrastructures and methods for a CDN, and for choosing a CDN service with different considerations.

Keywords: Content Delivery Network, Consistency Maintenance, Scalability.

1 INTRODUCTION

Over the past decade, Content Delivery Networks (CDNs) have seen a dramatic increase in popularity and use. There were 28 commercial CDNs [1] reported in this crowded market, including Akamai, Limelight, Level 3, and more recent entrants like Turner and ChinaCache. Among them, Akamai [2], as a major CDN, has more than 85,800 servers in about 1,800 districts within a thousand different ISPs in more than 79 countries. The trend of scale is growing rapidly at about 50% per year, due to the 100% increase of traffic per year [3]. The vast growth of traffic and infrastructure illustrates that CDNs serve as a key part of today's Internet and undertake heavy content delivery load. This promising growth makes CDNs a hot spot for research. Figure 1 shows the standard architecture of current CDNs [4]. When an end-user tries to visit web content, the request is forwarded to the local DNS server, which returns the IP address of a content server if the IP exists in the cache and is not expired. Otherwise, the local DNS server forwards the request to the CDN's authoritative DNS servers, which return the IP address of the content server close to this end-user with load-balancing consideration [5]. Then, the user sends its content request to the IP address of a content server, which returns content. The content servers periodically poll content updates from the content provider. We use the server and provider to denote the content server and content provider in short, respectively.

- * Corresponding Author. Email: shenh@clemson.edu; Phone: (864) 656 5931; Fax: (864) 656 5910.
- Haiying Shen and Guoxin Liu are with the Department of Electrical and Computer Engineering, Clemson University, Clemson, SC, 29634. E-mail: {shenh, guoxinl}@clemson.edu
- Harrison Chandler is with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109. E-mail: hchandl@umich.edu
- Jin Li is with the Microsoft Research, Redmond, WA 98052. E-mail: jinl@microsoft.com

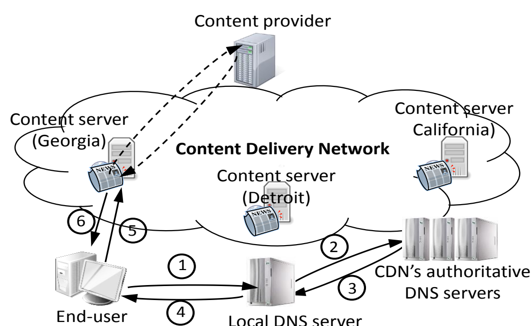


Fig. 1: The architecture of CDNs.

CDNs not only serve static contents without updates such as photos and videos, but also serve dynamic (or live) contents such as live game statistics, e-commerce and online auctions. The dynamic contents have frequent updates, which need to be delivered from providers to all replicas. Caching/replicating to surrogate servers near the network edge is widely used in CDNs to optimize the end user experience with short access latency. The large amount of widely scattered replicas make the consistency maintenance methods non-trivial. In addition, this method has two key requirements: scalability and consistency guarantee.

Based on the infrastructure, there are three common architectures used to deliver updates for consistency maintenance: i) unicast [6], [7], [8], [9], ii) broadcast [10] and iii) multicast tree [11], [12], [13], [14], [15], [16], [17], [18], [19], [20]. However, none of these approaches can satisfy both requirements simultaneously. The unicast approach can guarantee consistency, but since it relies on centralized content providers for updates, it causes congestion at bottleneck links and thus cannot promise scalability. Broadcasting can efficiently propagate the updates inside a local network and guarantee consistency. However, it generates very high overhead due to an overwhelming number of update messages. Thus, it cannot support the scalability required for large world-wide CDNs due to a vast number of redundant messages. The multicast approach produces fewer update messages than broadcasting, but node failures break the structure

1045-9219 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2479222, IEEE Transactions on Parallel and Distributed Systems

connectivity and lead to unsuccessful update propagation. Aside from node failures, the structure maintenance will incur high overhead and complicated management due to the dynamism of servers in the multicast tree.

With each update architecture, there are three basic methods for updating replicas: i) Time To Live (TTL) [6], [21], [8], [22], [18], [23], [9], [19], [24], ii) Push [12], [13], [14], [15], [16], [17], [20], [25], [26] and iii) server-based Invalidation [7], [9]. None of these update methods can guarantee the aforementioned two requirements. In TTL, servers poll the updates from providers whenever the TTL is expired, which supports greater scalability. TTL offers a tradeoff between freshness and CDN efficiency, and it can be dynamically changed based on update rates. In Push, an update is transmitted to every replica right after updating time, which guarantees a short period of inconsistency. However, an update will be pushed to all replicas immediately, which depends on the consistency infrastructure to support scalability. Also, Push may generate unnecessary update messages to uninterested replicas. In server-based Invalidation, whenever there are updates on content providers, an invalidation message is received by each replica, and replicas only fetch the update whenever the needed content is invalid. It can save traffic cost compared to Push if the content visit rates on servers in CDNs are smaller than the update rate of this content.

None of current update architectures together with update methods can fully solve both scalability and strong consistency in current CDNs. With the rapid growth of CDNs, consistency maintenance in CDNs needs to be particularly studied. Can the current update method used in the CDN provide high consistency for dynamic contents? If not, what are the reasons for the content inconsistency? What are the advantages and disadvantages of employing previously proposed consistency maintenance approaches in the CDN environment? *The answers to these questions help develop consistency maintenance approaches specifically for CDNs with different considerations.* Thus, in this paper, we focus on measuring the inconsistency of a CDN's servers, and break down the reasons for this inconsistency. Then, we conduct a trace-driven evaluation to measure the performance of consistency maintenance infrastructures and methods, and different parameters' effects on performance. The contributions of this paper are as follows:

- *Measuring the inconsistency of a major CDN.* This paper is the first to measure content consistency for a large amount of globally scattered servers in a major CDN. We measure the inconsistency of individual servers when delivering live sports game statistics.
- *Breaking down the reasons for the inconsistency of the CDN.* Through our measurement, we break down the reasons for inconsistency to different factors and analyze their effects. These factors include TTL value, propagation delay, shortage of bandwidth, content server overload/failure, content providers' inconsistency and so on.
- *Deducing the update infrastructure used in the CDN.* We check for the usage of static and dynamic multicast tree methods among clusters and servers to propagate the updates from the content provider to content servers, and confirm that the content servers directly poll updates from content provider based on unicast when serving a live game statistic content.
- *Evaluating infrastructures and methods for consistency maintenance through trace-driven experiments.* We further evaluate the performance for different infrastructures and update methods, since CDNs can easily adopt the other infrastructures with different update methods [27]. We itemize the advantages and disadvantages of different methods and infrastructures in different scenarios through our evaluation. We aim to give guidance for appropriate selections of consistency maintenance infrastructures and methods for a CDN or choosing a CDN service with different considerations.
- *Discussing hybrid and self-adaptive methods.* According to the features of the updates of the dynamic contents in the trace, we design a hybrid and self-adaptive method to save network load, respectively. The experiments validate the effectiveness of our design and show the promise of hybrid and self-adaptive methods.

The remainder of this paper is structured as follows. Section 2 presents a concise review of related works about the update methods and infrastructures. Section 3 summarizes the consistency measurement of dynamic contents in a major CDN and analyzes the trace to propose our findings. Section 4 evaluates the advantages and disadvantages of different update methods and infrastructures through trace driven experiments. Section 5 proposes two hybrid update methods and validates their effectiveness. Section 6 summarizes this work and proposes future work.

2 Related Work

Commercial CDNs enable efficient delivery for many kinds of Internet traffic, such as e-commerce and live sports. Serving dynamic contents not only requires a scalable CDN, but also requires consistency guarantees, either strong or weak. Recent studies of consistency maintenance have been applied to different applications, such as P2P networks, web caches, and CDNs. Based on the infrastructure, these studies can be categorized into three classes.

One class of methods is based on unicast. In [7], an Invalidation method is recommended, since it is better at saving traffic costs and reducing end-user query times. In [25], [26], [28], [29], [30], Push methods are used for consistency maintenance. Tang *et al.* [8] analyzed the performance of TTL-based consistency in an unstructured P2P network and studied the impact of consistency with different values of TTL. In [6], [22], [24], an adaptive TTL is proposed to predict the update time interval based on a historical record of updates. In [9], a hybrid update method is proposed, which depends on Invalidation to notify of outdated data and then uses an adaptive TTL method to poll for an update. Compared to a fixed TTL [21], the adaptive TTL may reduce traffic costs as well as support stronger consistency. However, the modification behavior of a content is not natural, which is hard to predict to guarantee consistency.

Another class of methods is based on broadcasting. Lan *et al.* [10] proposed to use flooding-based Push for near-perfect fidelity or a push/pull hybrid method for high fidelity. Broadcasting is widely used in local computer networks but fails to be sufficiently scalable for use in large scale networks such as CDNs due to a large number of redundant messages.

The last class of methods is based on multicast. In [13], an application-level multicast infrastructure is adopted to push updates to a group of servers serving the same cached data content in a CDN. In [20], a Push-based method is used to ensure strong consistency of all the metadata, which validates the freshness of contents in information centric networking. Li *et al.* [17] presented a scheme that builds replica nodes into a proximity-aware hierarchical structure (UMPT) in which the upper layer form a DHT and nodes in the lower layer attach to physically close nodes in the upper layer. SCOPE [16] builds a replica-partition-tree for each key based on its original P2P system. It keeps track of the locations of replicas and then propagates updates. CUP [14] and DUP [15] propagate updates along routing paths. In FreeNet [12], updates are routed to other nodes based on key closeness. In a hybrid push/poll algorithm [11], flooding is replaced by rumor spreading to reduce communication overhead. When a new node joins or a node reconnects, it contacts online replica nodes to poll updated content. This hybrid push/poll scheme only offers probabilistic guarantee of replica consistency. GeWave [18] builds a poll-based multicast tree for consistency maintenance, in which the replica of a parent node has higher visit

1045-9219 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2479222, IEEE Transactions on Parallel and Distributed Systems

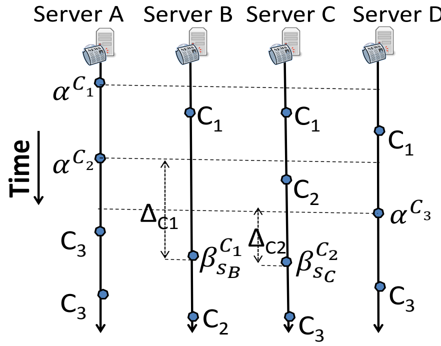


Fig. 2: An example of inconsistency.

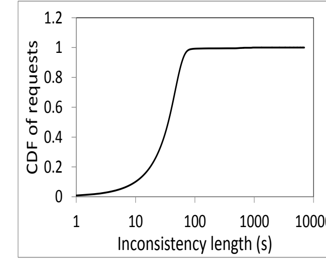


Fig. 3: The inconsistencies of data served by the major CDN.

frequency than the replicas of their child nodes. Tang *et al.* [19] proposed a method to reduce consistency maintenance costs and maintain low access latencies via an optimal replica placement scheme in CDNs, which leverages a TTL-based method to poll updates through a multicast tree. Tang and Zhou [31] studied inconsistency in distributed virtual environments, which create a common and consistent presentation of a set of networked computers. Peluso *et al.* [32] introduced a distributed multi-version concurrency control algorithm for transactional systems that rely on a multicast-based distributed consensus scheme. S2PC-MP [33] is a metadata consistency scheme for distributed file systems based on multicast. Benoit *et al.* [34] studied replica placement in tree networks subject to server capacity and distance constraints, and proposed efficient approximation algorithms for the placement problem without considering the consistency maintenance. The work in [23] proposed a geographically-aware poll-based update method. It builds a tree and let children poll parents to get the updates.

So far, there has been no consistency maintenance method specifically proposed for large-scale CDNs. Also, there has been no study that investigates the content inconsistency in current CDNs based on real trace. This is the first work that analyzes the consistency performance and causes in a major CDN based on the real trace, and extensively evaluates the consistency and overhead performance in trace-driven experiments in different scenarios.

3 Trace Analysis

3.1 Measurement Methodology

In order to study the consistency maintenance strategies used in current CDNs, we crawled cached content of a popular sports game from a large number of servers in a major CDN. The content we crawled was live game statistics webpages that need to be continuously updated throughout the game. To identify the IP addresses of the CDN content servers, we retrieved all domain names in all webpages, and used the method in [4] to translate the domain names to IP addresses by using their local DNS servers. Then, we validated each IP address's corporation to derive the IP addresses of the content servers and providers using the same method in [4]. Finally, we found 10 provider IP addresses and 50064 CDN IP addresses. Compared to the IP addresses of the CDNs crawled in [4], we have crawled most (57.2%) of the IPs in [4], which has 59581 IPs in total. There are 26.9% more new servers compared to their trace, which indicates the rapid scale increase of the CDN. By looking through the webpage source codes, we found out the content provider's domain name. Then, we also used the same method to track the IP addresses of the content provider's origin servers.

We randomly selected 200 globally distributed lightly loaded PlanetLab nodes. Then, we randomly selected 3000 content servers, each of which can continuously respond the content requests with low latency to one of the PlanetLab nodes. Each content server had one PlanetLab node simulating an end-user to poll live game statistics from it every 10 seconds for two and half hours each day. We collected 15 days of trace data between May 15, 2012 and June 4, 2012.

To measure data inconsistency, for each poll we retrieve the snapshot of statistics and current GMT (Greenwich Mean Time) time on that content server in order to avoid the interference of network delay. However, the GMT time may not be synchronized among all content servers; we use a method to remove the influence of the lack of synchronization. Specifically, we randomly chose a PlanetLab node n_i and then adjust each server's time to the time in n_i. To achieve this, we make node n_i poll the content from each content server s_j, and the GMT time difference of server s_j (denoted by ϵ_{n_i,s_j}) is calculated by $t^G_{s_j} - t^G_{n_i} - RTT/2$, where $t^G_{n_i}$ and $t^G_{s_j}$ are the GMT times on PlanetLab node n_i when starting the query and on s_j upon receiving the query, respectively, and RTT is the average round trip time of a query between n_i and s_j. Then, we subtract ϵ_{n_i,s_j} from the GMT time associated with each snapshot from each server s_j in order to make the timestamps of all snapshots be consistent with the GMT time on n_i. As shown in Figure 2, we identified different snapshots from all polled snapshots and use C_i to denote the ith content snapshot. We find the first time when each snapshot C_i shows up in the trace and denote it as α^{C_i}. For each server s_n, we ordered its content snapshots over time. For each C_i, we find the last time that C_i shows up, which is denoted by $\beta^{C_i}_{s_n}$. The inconsistency length of C_{i-1} (denoted by $\triangle_{C_{i-1}}$) for that server is calculated by $\triangle_{C_{i-1}} = Max\{\beta^{C_{i-1}}_{s_n} - \alpha^{C_i}\}$. It denotes that a server first saw the updated data C_i at time α^{C_i}, and another server lastly saw the unupdated data at $\beta^{C_{i-1}}_{s_n}$, indicating that the inconsistency existed for time length $\triangle_{C_{i-1}}$. Since we poll the contents from a very large number of servers, the first time an update is observed should be close to the time of this update at the content provider. The inconsistency length of t seconds means that the content is expired for at least t seconds.

3.2 Inconsistency in the Content Servers

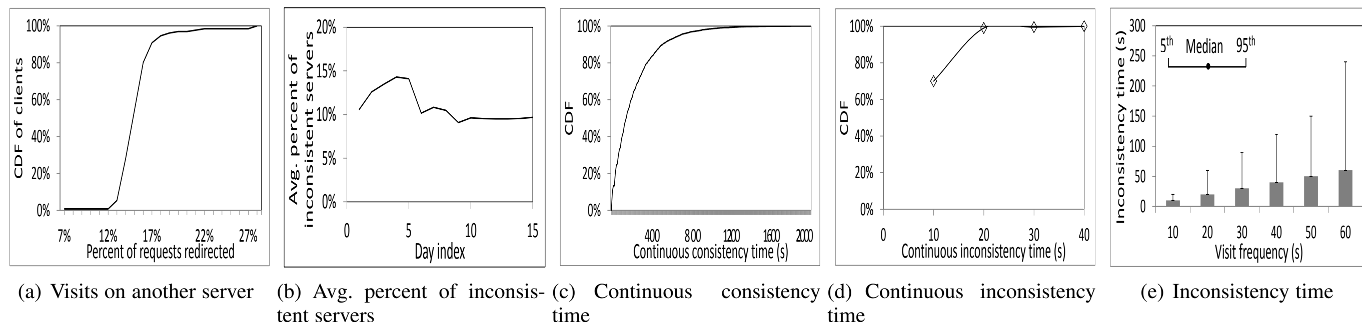
Figure 3 shows the cumulative distribution function (CDF) of inconsistency lengths for all content requests during the 15 days. We see that only 10.1% of requests have inconsistency lengths less than 10 seconds, and 20.3% of requests have inconsistency lengths greater than 50 seconds. The results indicate that content inconsistency exists among the content servers in serving the dynamic contents of game statistics that require frequent updates.

3.3 Inconsistency Observed by Users

The game statistics are sequenced over time. A user observes self-inconsistency (inconsistency in short) when (s)he sees statistics (such as game score) prior to the most recently statistics (s)he has seen. For example, a user sees a game score of 2:3 at 1:00pm, and then sees the score changes to 2:2 at 1:01pm. Such user observed inconsistency is caused by the user receiving older webpage content that is not updated in time from another server.

1045-9219 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2479222, IEEE Transactions on Parallel and Distributed Systems



(a) Visits on another server (b) Avg. percent of inconsistent servers (c) Continuous consistency time (d) Continuous inconsistency time (e) Inconsistency time

Fig. 4: User perspective consistency.

The local DNS server caches the IP of the CDN's server for each visited domain name. The IP expires in a short time. When the local DNS server receives a request from an end-user, if the IP is not expired, it sends the IP to the end-user; otherwise, it forwards the request to the authoritative DNS servers for the IP of a server. The authoritative DNS servers consider the load balancing between the servers and send back an IP of a server. To update the dynamic contents in a webpage, the end-user sends out a request every 10 seconds. Therefore, the request may be redirected to another server due to the expired cached IP in the local DNS server and the server reassignment by the authoritative DNS servers. If the content in the newly assigned server is not updated, the user may observe the inconsistency.

To investigate user-observed inconsistency, we used 200 world-wide distributed PlanetLab nodes to visit the same game statistics through its URL once every 10 seconds during the time period of a game. We recorded the IPs of serving servers and the received statistics contents for each user. We measure the consistency performance from a single user's perspective. First, we introduce three metrics: the percent of requests redirected to another server, continuous consistency time and continuous inconsistency time.

Suppose a user creates N requests in total, in which M requests are served by redirected servers; then, the percent of requests redirected to another server equals $\frac{M}{N}$. Continuous inconsistency time is the period of time between when the user observes an inconsistency to the time of next consistency record. Continuous consistency time is the period of time between when the user observes a consistent record to the time of next inconsistency record.

We first measure the percentage of a single user's visits that are redirected to a server different from the current server. Figure 4(a) shows the CDF of users versus the percent of visits redirected to another server. It shows that most of the users have 13%-17% of visits switched to another server. From the trace we observed that with continuous updates of the game statistics, on average, there are around 11% of servers with inconsistent content at each polling time during all 15 days as demonstrated in Figure 4(b), which shows the average of percent of inconsistent servers in every 10 second on each day. This means that on average, a user's request has around 11% probability of being redirected to a server that has outdated content. Thus, 1.43% to 1.87% of visits of a single user during a game will be redirected to outdated content.

We then measure the continuous (in)consistency time from a single user's perspective. We calculated all continuous (in)consistency times of all users with a request redirection probability as shown in Figure 4(a). Figure 4(c) shows the CDF of the continuous consistency time. The median continuous consistency time is around 160 seconds, and 82.4% of all continuous consistency times are within 400 seconds. The result means that most users can observe inconsistency and receive outdated contents during watching. Figure 4(d) shows the CDF of continuous inconsistency times. In this figure, 70% of all the continuous inconsistency times are 10 seconds or less, and around 99% of all the continuous inconsistency times are no longer than 20 seconds. There are no inconsistency times longer than 40s. The result indicates that users may observe outdated dynamic web content, but it always lasts no more than 20 seconds, which means that the inconsistency usually lasts no more than two continuous visits.

We varied the polling frequency from 10 seconds per poll to 60 seconds per poll with a 10 second increase in each step. The 95th percentile measurement is widely used to calculate the network bandwidth usage [35]. We adopt the 95th percentile measurement to evaluate the inconsistency in this paper. For each polling frequency, we collected all continuous (in)consistency times of all users and calculated the 5th percentile, median and 95th percentile of the continuous inconsistency times. Figure 4(e) shows the results with different polling frequencies. From the figure, we observe that the median value always equals the 5th percentile value, meaning most inconsistency lasts for a short time period. Also, we see that the median and the 95th percentile value of the inconsistency time increase in proportion to the visit frequency due to the slower polling frequency. From Figure 4 and Figure 4(e), we can infer that an individual user can observe inconsistency on dynamic contents in the CDN. This implies that the current update strategy in the CDN can be improved to prevent users from receiving outdated information for dynamic contents.

3.4 Causes of Inconsistency in CDNs

In the previous section, we observed that content inconsistency exists in the servers and that end-users can also observe the inconsistency (i.e., receive outdated content). In the following, we identify and explore potential causes of content inconsistency in the CDN, which will provide guidance for designing consistency maintenance mechanisms. We measured the individual influence on the inconsistency among content servers of each potential cause, including the TTL value, content provider's inconsistency, provider-server propagation delay, content provider bandwidth shortage, content server overload/failure.

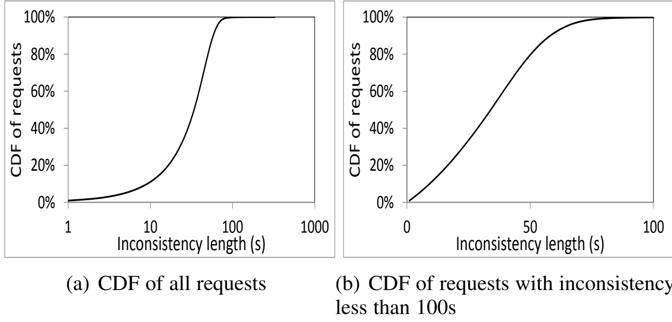
3.4.1 *Time-to-live based consistency maintenance*

In the TTL-based consistency maintenance method, when a content server receives a content request from an end-user, it first checks its cache for the content. If the content exists in the cache and its TTL has not expired, the server serves the content. If the content does not exist in the cache or its TTL has expired, the server retrieves the content from the content provider, sends it to the user, and caches the content. Therefore, if the content changes before the TTL has expired, the content server will inadvertently fulfill requests with outdated content. [36] indicates that the CDN uses TTL-based consistency maintenance.

We study the impact of the TTL-based method on content inconsistency. In order to minimize the influence of the provider-server distance and server-user distance on content inconsistency, we clustered geographically close servers, used the same or geographical close PlanetLab nodes to poll the contents from

1045-9219 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

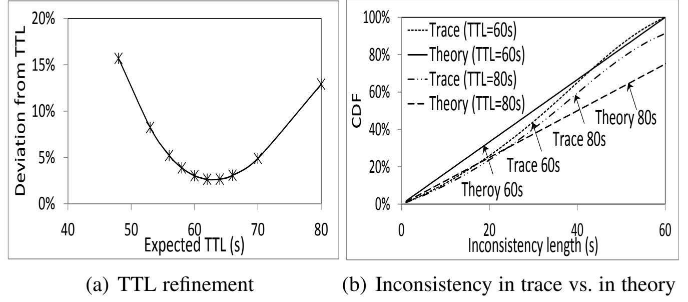
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2479222, IEEE Transactions on Parallel and Distributed Systems



(a) CDF of all requests

(b) CDF of requests with inconsistency less than 100s

Fig. 5: CDF of inner-cluster inconsistency.



(a) TTL refinement

(b) Inconsistency in trace vs. in theory

Fig. 6: The CDN content severs' TTL.

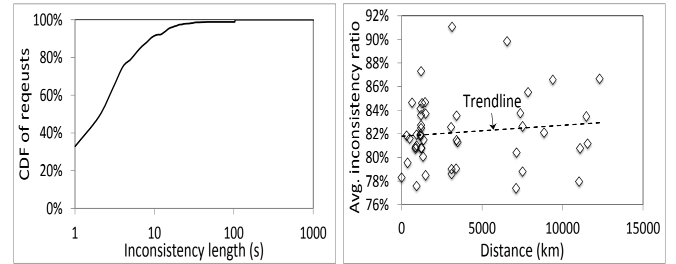


Fig. 7: The inconsistencies of data served from the provider.

Fig. 8: The distance of servers.

servers in the same cluster. We examined the distribution of inner-cluster inconsistency lengths, which refer to inconsistency lengths calculated only within clusters of collocated nodes rather than all servers in the CDN. To create clusters, we first translated the IPs of the CDN's servers to geographical locations by an online IP geolocating service [37], and grouped the servers with the same longitude and latitude into a cluster.

Figure 5(a) shows the CDF of requests for different inconsistency lengths. The figure shows that only 31.5% of served requests have inconsistency lengths less than 10 seconds. Also, the CDF of requests approximately exhibits a linear increase when the inconsistency length increases from 0 to 60 as shown in Figure 5(b), which plots the CDF of requests with inconsistency less than 100s. We can assume that the inner-cluster inconsistency length is evenly distributed in [0, TTL], which will show a linear increasing in CDF within [0,TTL]. Then we can assume that the TTL for cached content is around 60 s.

Below, we attempt to derive TTL using another method. We assume all updates are independent, and all servers independently start to cache the dynamic contents. We then derive the average inconsistency lengths of all servers, denoted by $E[I]$. If we split the time into slots, each of which lasts TTL, then a server can poll the content at any time within $[0, TTL]$ in a slot with the same probability. If the first server gets the update C_i at time t, since servers poll the content independently, other servers receive the update at any time t' within $[t, t+TTL]$ with a uniform distribution. As the inconsistency length equals $t'-t$, it is then uniformly distributed within $[0, TTL]$. Thus, $E[I] = \frac{TTL}{2}$.

Since TTL is not the sole factor of inconsistency, the true average inconsistency length from the trace denoted by $E'[I]$ is larger than $E[I]$. If TTL is the sole factor, $2E'[I]$ should not be larger than TTL. Therefore, we use recursive refining to derive the TTL used by the CDN from the trace. We first calculate the average inconsistency length in trace $E'[I]$, and then calculate $TTL' = 2E'[I]$. Then, we calculate $E''[I]$ from the inconsistency lengths in the trace that are no larger than TTL' and derive a new $TTL'' = 2E''[I]$. We then calculate the deviation of the two TTLs as $(TTL'' - TTL')/TTL'$. We repeat this procedure to derive a TTL, which is closest to $2 * E[I]$ according to the above equation. Thus, the TTL' with the smallest deviation is the actual TTL used in the CDN. Figure 6(a) shows the deviation distribution versus each derived expected TTL. The smallest deviation is approximately at 60s, which means that we can infer the CDN's TTL to be 60s.

We then verify if TTL=60s is correct. Using 60s and 80s as the TTL respectively, we calculate the CDF of the inconsistency length. For TTL=60s (and 80s), we remove the inconsistency lengths larger than 60s (and 80s) in the trace (which are inconsistencies caused by reasons other than the TTL) and plotted the inconsistency distribution based on the remaining data. Figure 6(b) shows the CDF of inconsistency lengths. From the figure, we see that the deviation between the trace and theoretical inconsistency with TTL=60s is smaller than that with TTL=80s. The root mean square error of TTL=60s is 0.0462, while that of TTL=80s is 0.0955. We also test other TTL values, and find that TTL=60s leads to the smallest deviation between trace and theoretical inconsistency. Thus, the actual TTL should be 60s, which introduces an average inconsistency length of 30s. We notice that the average inconsistency is 40s in the trace. Thus, we can conclude that TTL is the main cause for the inconsistency, and other factors such as provider/server inconsistency, content server failure and overload, network congestion introduce a small part of inconsistency, around $\frac{80-60}{80} = 25\%$.

3.4.2 Content provider inconsistency

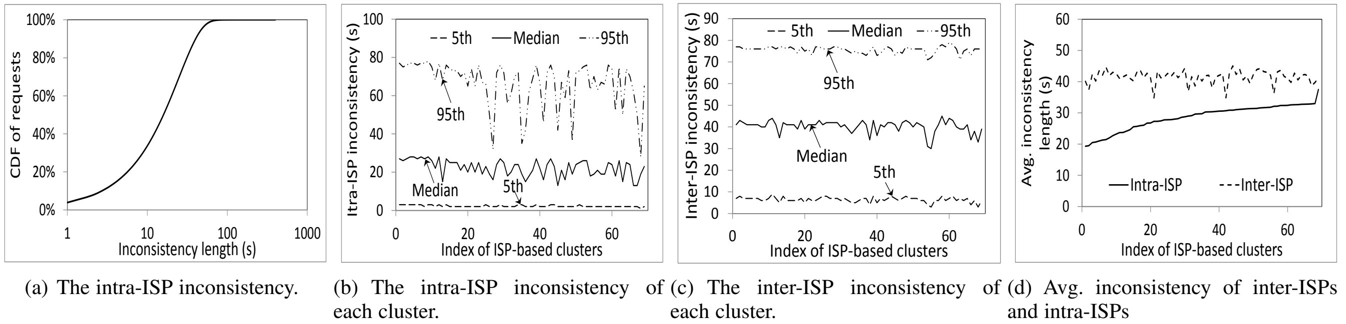
One potential cause of inconsistency between content servers is inconsistency at providers that provide contents to the content servers. We requested statistics contents for the same game from the providers using the same setup as before. Figure 7 shows the CDF of inconsistency length for requests served by the providers. The figure shows that 90.2% of served requests have inconsistency lengths less than 10 seconds, only 1.2% of requests have inconsistency lengths greater than 50 seconds, and the average inconsistency is 3.43 seconds. The inconsistency length is much lower than that of the CDN-served content as shown in Figure 3. We have checked the geographical locations of all our identified providers and found that they are in the same geographical location. In this case, the content providers can provide higher consistency than the servers that are dispersed worldwide. Even if multiple providers deliver the same dynamic content to the servers, the providers have negligible content inconsistency; therefore, their responsibility for the inconsistent contents received by end-users is negligible.

3.4.3 Provider-server propagation delay

Content servers are distributed globally so that end-users can be served by their geographically close servers. However, globally dispersed servers face considerable propagation delay for content originating from a central location. Since propagation delay varies for different servers, inconsistency can be introduced. We introduce a metric called *consistency ratio* for a server, which is calculated by $1 - \frac{\sum inconsistency\ lengths}{total\ trace\ time}$, which indicates the capability of a server to maintain consistency. We clustered servers with the same distance to the provider and calculated the average consistency ratio.

1045-9219 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2479222, IEEE Transactions on Parallel and Distributed Systems



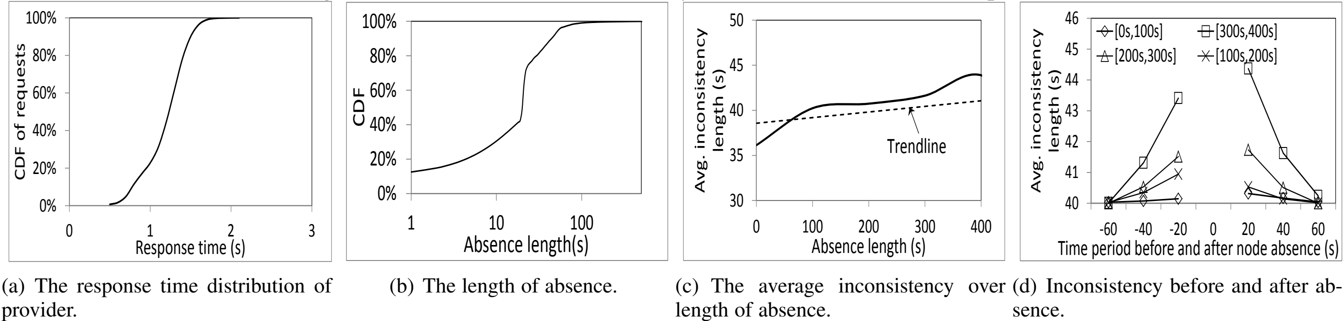
(a) The intra-ISP inconsistency.

(b) The intra-ISP inconsistency of each cluster.

(c) The inter-ISP inconsistency of each cluster.

(d) Avg. inconsistency of inter-ISPs and intra-ISPs

Fig. 9: The effect on the inconsistency of inter-ISP traffic from the provider.



(a) The response time distribution of provider.

(b) The length of absence.

(c) The average inconsistency over length of absence.

(d) Inconsistency before and after absence.

Fig. 10: The effect of server overload/failure on the inconsistency.

Figure 8 shows the average consistency ratio versus the provider-server distance. The figure shows that in that overall, as the provider-server distance increases, the average consistency ratio exhibits a very slight increase. The average consistency ratio and distance have little correlation ($r = 0.11$). As distance is directly related to propagation delay, this result indicates that propagation delay has a little effect on inconsistency.

We further investigate whether inter-ISP traffic has an effect on inconsistency. Traffic transmitting between ISPs is more costly for ISPs, and such traffic competes for the limited transmission capacity [38]. To identify the ISP for each server, we first found the ISP of each server based on its IP using IPLOCATION [37]. We further increase the accuracy of these identified ISPs. We use Traceroute to diagnose the entire path for each request from a PlanetLab node to a content server. Since the CDN's servers are close to the backbone routers of ISPs [4], we checked whether the router in the last several routing hops in a route belongs to the identified ISP. If not, we removed the trace record of the server. We successfully verified the ISPs of 99.6% of the servers.

We grouped servers within the same ISP to a cluster. We calculated the inconsistency lengths for all servers in each ISP-based cluster. Figure 9(a) shows the CDF of the intra-ISP inconsistency of all clusters. We see that 33.7% of requests have inconsistency lengths less than 10 seconds, and only 3.9% of requests have inconsistencies greater than 60 seconds. This inconsistency distribution is only slightly better than that in Figure 3. Thus, we can conclude that although inter-ISP traffic competes for the transmission capacity, it only contributes slightly to inconsistency in servers on average.

In order to better understand the degree of influence of inter-ISP traffic from providers on the inconsistency, we compare the intra-ISP and inter-ISP inconsistency lengths. The inter-ISP inconsistency lengths are calculated using the same method as the intra-ISP inconsistency lengths except that the α^{C_i} of a cluster is the earliest time of C_i's appearance in all other clusters. Figures 9(b) and 9(c) show the 5th, median and 95th percentiles of the intra-ISP and inter-ISP inconsistency lengths of each ISP-based cluster. We see that the inter-ISP inconsistency lengths are always higher than the intra-ISP inconsistency lengths, meaning the inter-ISP traffic from providers affects the inconsistency. The median percentiles of the intra-ISP and inter-ISP inconsistency lengths range from [13,28] and [30,45] respectively, and the 95th percentiles of the intra-ISP and inter-ISP inconsistency lengths range from [36,73] and [71,79] respectively. The median and the 95th percentiles of the inter-ISP inconsistency lengths are larger than those of the intra-ISP inconsistency lengths, and the increment indicates the degree of influence of inter-ISP traffic from providers. The increment of the average inconsistency lengths is illustrated in Figure 9(d). We see that on average, the inter-ISP traffic from providers increases the inconsistency lengths by [3.69, 23.2]s.

3.4.4 *Content provider bandwidth*

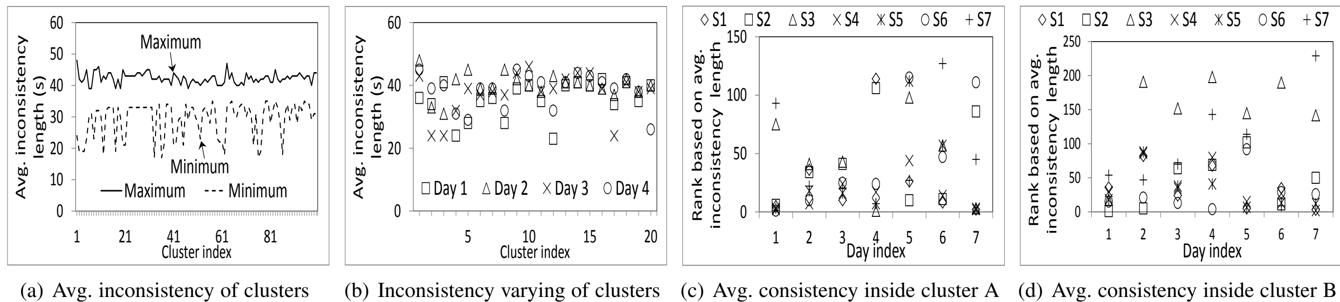
If the providers are overloaded or have insufficient bandwidth, they will not be able to receive up-to-date content. We measure each request's response time by $t_r - t_i$, where t_i is the time that the PlanetLab node initiates the request and t_r is the time that it receives the content. Figure 10(a) plots the CDF of requests versus the response time. The figure shows that the response time are in the range of [0.5, 2.1]s, and 90% of requests are resolved within 1.5s. Thus, there is no large delay due to congestion or overloaded of the providers. This indicates that the content providers have sufficient computing capabilities and bandwidth to handle all requests. Considering the time interval between the worst case and the best case of the response time, the provider's network resource constraint only introduces less than 1.6s inconsistency in servers. In the trace, the size of content is relatively small, so it hardly causes congestion in provider's uplinks. However in some situations, such as live video streaming, the content provider's bandwidth may introduce large inconsistency when overloaded.

3.4.5 *Content server failure and overload*

Content server failure and overload could also be the cause of content inconsistency. When a server has failed or is overloaded, it cannot quickly send out content requests to or receive content from the provider. If the IPs of failed or overloaded content servers are cached in local DNS, end-users will acquire these cached IPs from the local DNS service and observe inconsistent contents from these servers. Suppose that two successive response times of a server upon polling are t_i and t_{i+1}. We calculated the *absence*

1045-9219 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2479222, IEEE Transactions on Parallel and Distributed Systems



(a) Avg. inconsistency of clusters (b) Inconsistency varying of clusters (c) Avg. consistency inside cluster A (d) Avg. consistency inside cluster B

Fig. 11: Static multicast tree non-existence.

length of the server as $t_{i+1} - t_i - 10s$, where $10s$ is the time interval of two successive polls. The absences could be due to node overload, reboot, or failure. Suppose the content responded at t_{i+1} from the content server that was absent is C_{i+1}, then we call the inconsistency length of C_{i+1} *the inconsistency length of this absence*.

Figure 10(b) shows the CDF of absence lengths of servers. The figure shows that absence lengths range from [1,500] seconds, with 30.4% less than 10 seconds and 93.1% less than 50 seconds. The CDN has a load balancing technique that balances the load between servers, so it is unlikely that an overloaded node would remain in the overloaded status for an extended period. Thus, node failures/reboots are responsible for most of the absences lasting longer than 50 seconds.

We plot the node absence length with the average inconsistency length after node returns in Figure 10(c). We group trace records first by their absence length. Since for a specific absence length, there may not be enough inconsistencies to show its general case of inconsistency with such an absence length, we group absence length by every 50s. In order to show the average inconsistency without absence, we divide first group with absence length of [0,50]s to [0,0]s and (0,50]s. From the figure, we can see that the inconsistency is increasing slightly from 38.1s to 43.9s while the absence length increases from 0s to 400s. It indicates that the node overloads and failures have adverse effects on consistency. They can increase the average inconsistency by 15.22%.

In order to determine the effect of node absences on the inconsistency, we grouped the inconsistency lengths associated with the same absence length and calculated the average. We then group the average absence lengths with inconsistency lengths in the range of (0,50]s, (50,100], $\cdots$, (350,400]. In order to show the average inconsistency without absence, we also plot the average inconsistency lengths for absence length equals 0. Figure 10(c) plots the average inconsistency length in different ranges of absence lengths. From the figure, we see that the inconsistency length increases from 38.1s to 43.9s while the absence length increases from 0s to 400s. That is, the absence contributes no more than 6s inconsistency. This indicates that the overload and failure do have influence on consistency and that they can increase the average inconsistency by 15.22%. The figure also shows that a larger absence length leads to a higher average inconsistency length, because servers may not receive updates or send out update requests in time when overloaded or failed.

Recall that the inconsistency length of C_{i-1} is calculated by $\triangle_{C_{i-1}} = Max\{\beta_{s_n}^{C_{i-1}} - \alpha^{C_i}\}$. If a server's absence length is larger than 0, we calculated the inconsistency lengths of C_{i-1} polled during $[t_i - x, t_{i+1}]$ and $[t_i, t_{i+1} + x]$, where $x = 20, 40, 60$. For all the inconsistency lengths in each range of all absent servers, we derived those with absence lengths within [0s,400s], classified them to 4 groups with absence lengths in [0s,100s], [100s,200s], [200s,300s] and [300s,400s], and then calculated the average inconsistency length in each group. Figure 10(d) plots the average inconsistency lengths in each group in a certain time period before and after node absence. It shows that in each group, the inconsistency measured closer to the absence is larger and vice versa. We suspect this is because when a server is about to be overloaded or has just recovered from overload or failure, it has a lower probability of sending or receiving update requests. From the figure, we also see that a larger absence length leads to a higher inconsistency length and that sharper inconsistency length increase. These results indicate that server overload and failure affect inconsistency and we need to avoid system failure when there are continuous updates of the dynamic contents, as it causes largely degraded user experience.

3.4.6 Summary of the Causes of Inconsistency

We summarize the impact of different causes of content inconsistency in the following. We order the different causes based on the strength of their impacts.

- TTL is the main cause of the content inconsistency and contributes 75% of the content inconsistency.
- Content server failures and overloads can increase the average inconsistency by 15.22%. Also, a larger absence length leads to a higher average inconsistency length.
- Provider-server propagation delay has a small effect on content inconsistency. The main cause is the inter-ISP traffic of providers, which increases the inconsistency lengths by [3.69, 23.2]s.
- The content provider's network resource constraints introduce less than 1.6s inconsistency in servers.
- Content provider inconsistency is low and introduces negligible effect on content inconsistency.

3.5 Multicast-Tree Existence

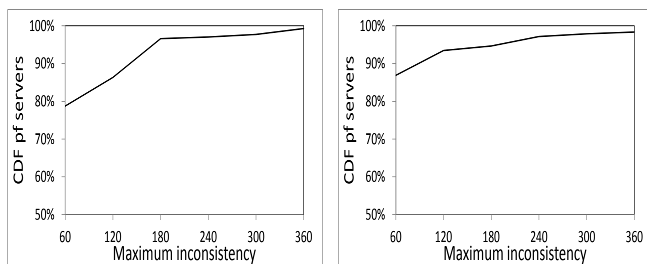
3.5.1 Static Proximity-Aware Multicast Tree Existence

In this section, we analyze the base infrastructure that the CDN uses to distribute dynamic content updates. Recall that a server polls the provider for an update every TTL. We first attempt to determine whether there is a static proximity-aware update tree to update the dynamic contents. The proximity-aware multicast tree is widely used to propagate updates to all replicas in order to reduce the network load and improve scalability [17], [18], [39]. Due to the proximity-awareness, we can assume that the servers geographically close to each other and within the same ISP should be in the same cluster.

To infer the existence, we must check whether there is a tree structure among clusters to update the contents. We calculated the average inter-cluster inconsistency length of each cluster for each day during the 15 days. Figure 11(a) plots the minimum and the maximum average inconsistency of each cluster of one day. Due to limitations of the plot area, we only show the inconsistencies of 100 randomly selected clusters; however, all the other clusters have similar results. We see that the average inconsistency of any cluster length varies greatly during the trace. If there is a static proximity-aware tree structure among clusters, the inconsistency of the cluster in the higher layer of this tree should always have smaller inconsistency. Figure 9(c) shows a similar tendency of inter-cluster inconsistency as shown in Figure 11(a), where the servers are grouped by their ISPs into clusters. Neither figure

1045-9219 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2479222, IEEE Transactions on Parallel and Distributed Systems



(a) CDF of maximum inconsistency in Day A (b) CDF of maximum inconsistency in Day B

Fig. 12: Dynamic multicast tree non-existence.

shows a tiered layer of inconsistency among clusters, in which clusters with smaller inconsistency should always have smaller inconsistency. Thus, the figures indicate that there is not a static proximity-aware tree among clusters for content updates. Further, we measure the average inconsistency of each cluster for 4 successive days as shown in Figure 11(b). Though all clusters together have similar results, due to limitation of the plot area, we randomly select 20 clusters to be shown in the figure. In the figure, each marker stands for one day. From the figure, we can see there is no static inter relationship among clusters. Since we randomly selected the content servers geographically distributed over the world to crawl the trace,we can infer that the clusters are at different layers of the tree. In a multicast tree, the higher layers of the tree should always have smaller inconsistency lengths than the servers in the lower layers. Then, the relative differences of the average inconsistency lengths of different clusters should remain similar in the 4 days, which contradicts the results as shown in Figure 11(a). Thus, it confirms that there is not proximity-aware tree structure among the clusters.

Further, we analyze whether a tree is used to distribute the updates between geographically close servers inside a cluster. We randomly selected two clusters for the analysis, denoted by Clusters A and B. Clusters A and B have 140 and 250 servers, respectively. For each cluster, we calculated the average inconsistency length of each server for each day and order the servers based on the average value for 7 successive days. The server with the lowest inconsistency length has rank 1. We randomly selected 7 servers from each cluster, and plot each server's rank across 7 successive days in Figures 11(c) and 11(d) for Clusters A and B, respectively. Recall that we use s_n to denote server n. We see that the rank of each server varies greatly. If there is a static tree structure among geographically close servers, the rank of each server should vary within a limited range, which contradicts the results as shown in the two figures. Thus, we can infer that most likely, a static tree structure is not used to distribute the dynamic web content updates among geographically close servers in the CDN.

3.5.2 Dynamic and Static Multicast Tree Existence

Then, we look for evidence of a multicast tree with proximity-awareness among servers. In the trace crawling, for each content server we continuously polled the content every 10 seconds as in Section 3.1. Therefore, we can assume that the dynamic multicast tree structure was unchanged during our polling. In order to remove the effect of dynamism of tree structure, we removed the trace of all servers with any absence. We measure the maximum inconsistency of each server for each day in the trace. The maximum inconsistency of servers at the second layer in the multicast tree (the root is the content publisher) should be upper bounded by TTL. Recall that we have derived that the TTL for each content server as 60 seconds. If there is a multicast tree to update the content, there are more nodes at lower layers than at the second layer. Recall that we randomly selected the content servers which are geographically distributed all over the world.

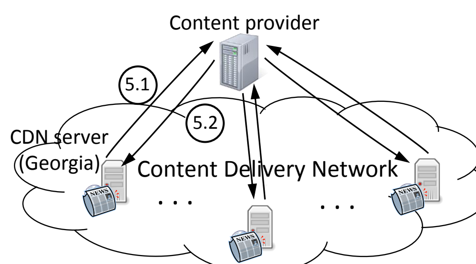


Fig. 13: The major CDN's architecture.

Therefore, we can infer that the probability that a content server in the trace is at a certain layer of the multicast tree is proportional to the number of content servers at that layer. Thus, our trace show that most of servers are at higher layers. Figures 12(a) and 12(b) show the CDF of maximum inconsistencies of all servers for Days A and B, respectively. They show that the majority of the servers have inconsistency less than 60 seconds, which are 76.7% and 86.9%, respectively. The same phenomenon is shown in the CDF of maximum inconsistencies of all servers on each of the other days in the trace. Since there are more servers with maximum latencies less than 60 seconds, that is contradict to the existence of a multicast tree, we can derive that there is no multicast tree among the content servers to propagate the updates with high probability. In all, we can infer that, when serving the dynamic contents in the trace, *a content server directly polls updates from the provider without a multicast tree.*

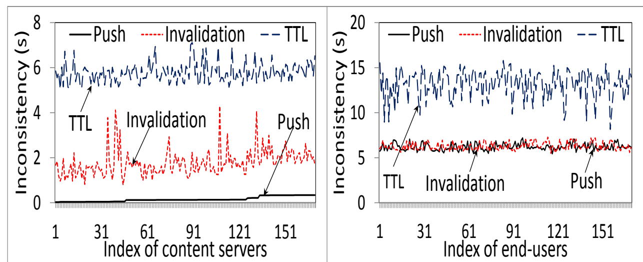
3.6 Summary of Trace Analytical Results

Finally, we can complement Figure 1 with the dynamic content update method and infrastructure between the content servers and the content provider with the analysis above. We can induce that the CDN depends on TTL for updates using unicast infrastructure, in which the content servers directly poll contents from the content provider. Therefore, when the user makes a request to the content server for content at Step 5 as shown in Figure 1, if the content on this server is missing, the server will directly visit the content provider using a http request for the new content as Step 5.1 as shown in Figure 13. Then, the content will be returned to the CDN's server at Step 5.2. After requesting the content, the content server will set a TTL for this live content. If the content TTL has expired, the server must retrieve the content from the content provider again. Thus, if there are consecutive visits towards this content on this server, it will repeat Steps 5.1 and 5.2 for consistency maintenance. Finally, the content will be returned to the client at Step 6 as shown in Figure 1.

From the analysis, we can see that the major CDN's servers indeed have large inconsistencies for their cached dynamic contents. Thus, the CDN cannot guarantee consistency for dynamic contents by using unicast with TTL method. Also, a user can observe the inconsistency of his/her viewed content due to server redirection. The inconsistency is caused by several factors including TTL, provider-server propagation delay, providers' inconsistency and bandwidth, server overload and failure. The biggest factor is the TTL, which contributes around 75% of average inconsistency. The other factors contribute to the inconsistency significantly less than TTL, and they are not easy or expensive to solve. For example, the server overload problem can be resolved by improving the capabilities of current servers and links. Compared to the other factors, improving the TTL-based consistency maintenance method is the easiest and only way to significantly improve content consistency. Further, the unicast infrastructure may cause congestion to the content provider and cannot promise scalability. We may need to deploy different infrastructure to relieve the load of central servers and improve the scalability for a more popular content

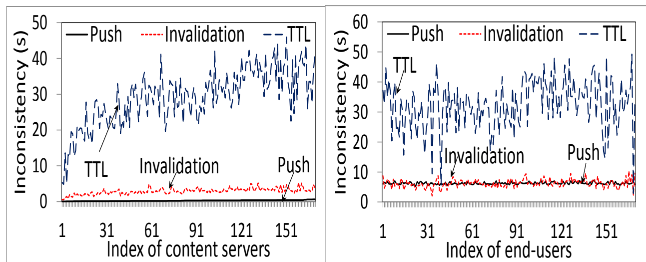
1045-9219 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2479222, IEEE Transactions on Parallel and Distributed Systems



(a) Content inconsistency of servers (b) Inconsistency of end-users

Fig. 14: Inconsistency in the unicast-tree infrastructure.



(a) Content inconsistency of servers (b) Inconsistency of end-users

Fig. 15: Inconsistency in the multicast-tree infrastructure.

with frequent updates. Thus, we conduct trace-driven experiments to evaluate the performance of different consistency maintenance approaches in a CDN to provide guidance for selecting or designing optimal consistency maintenance approaches for CDNs.

4 Trace-Driven Performance Evaluation

We conducted trace-driven experiments to evaluate the performance of consistency maintenance, scalability and overhead for different consistency maintenance infrastructures and methods in a CDN. The experimental results shed light on the selection or design of optimal consistency maintenance methods based on the varying needs of CDNs.

We built our simulated CDN on PlanetLab [40]. According to the distribution of the content servers in the CDN [4], we selected 170 PlanetLab nodes with high performance and light load mainly in the U.S., Europe, and Asia. We chose one node in Atlanta as the provider. We randomly selected one-day live game events on Jun. 2nd, 2012 in our trace data as the content. It includes 306 different snapshots lasting 2 hours and 26 minutes. We regard the time of each snapshot's first appearance in the trace as the update time in the provider. In each PlanetLab node, we also created five simulated end-users browsing the content.

We evaluate three different consistency maintenance methods, Push, Invalidation and TTL-based method (TTL in short), on two updating infrastructures, Unicast-tree and Multicast-tree. We do not evaluate broadcast, since CDN is a large network over various local networks, while broadcast is only effective and efficient inside a local network. In the multicast-tree, the provider is the tree root and geographically close nodes (measured by inter-ping latency) are connected to each other to form a binary tree. A larger d in $d-$ary tree leads to a smaller depth of the tree. Due to the small scale of the network, we chose $d = 2$ in order to emphasize the advantages and disadvantages of multicast compared to unicast. In the unicast-tree, the provider directly connects to all servers in multiple unicast channels. The size of all consistency maintenance related packages and content request packages were set to 1KB. According to the trace, the end users poll updates every 10s, and we call this polling time period end-user TTL. In each experiment, the provider starts to update contents at time 60s. Each end-user starts requesting the content from a time randomly chosen from [0s,50s].

4.1 Inconsistency in the Unicast-Tree Infrastructure

With the unicast-tree structure, in Push (or Invalidation), the provider directly sends updates (or notifications) to the servers, and in TTL, the servers poll the provider directly. With the multicast-tree structure, in Push and Invalidation, the update (or notification) is pushed along the tree in the top-down manner, and in TTL, the children poll their parents in the tree in the bottom-up manner. There are two layers of content inconsistency: i) the inconsistency between servers and the provider, and ii) the inconsistency between the end-users and the provider.

Figure 14(a) shows the average of all inconsistencies of each content server with different update methods in the unicast-tree infrastructure, where all servers are sorted by their inconsistency in Push. We see the inconsistency results follow Push<Invalidation<TTL. In Push, the provider pushes an update to the servers upon an update occurrence, leading to the smallest inconsistency. Its inconsistency is due to the traffic latency including the transmission delay, the propagation delay and the queuing delay at the output ports of the provider. In Invalidation, a server receives notifications for outdated content but does not request the new update until it receives a request from an end-user. Therefore, its inconsistencies are higher than Push. Since there are no user requests during the inconsistency time period, these inconsistencies do not affect the consistency of the contents received by users. In TTL, the content in a server is considered fresh during a TTL. This is why TTL generates the largest inconsistency, the average of which equals 5.7s, around $TTL/2$.

Figure 14(b) shows the largest average inconsistency of the end-users on each PlanetLab node. We see that Push and Invalidation produce similar inconsistencies that are lower than that of TTL. In Push and Invalidation, the servers always supply updated content. However, since end-users request the content periodically, they may send requests a certain time period after the content update, thus generating inconsistencies. This result implies that "pushing to end-users" should be a better method to improve the consistency of contents viewed by end-users. In TTL, the first-layer servers have certain inconsistencies. The second-layer content servers' periodical polling amplifies inconsistencies as they may not poll right after the servers' polling. Therefore, TTL generates larger inconsistencies than other methods. Also, TTL's inconsistencies of end-users are higher than those of the servers in Figure 14(b).

4.2 Inconsistency in the Multicast-Tree Infrastructure

In this section, we measure the inconsistencies of different methods in the multicast-tree infrastructure. Figure 15(a) shows the average of all inconsistencies of each server with different update methods in the multicast-tree infrastructure, where all servers are sorted by their inconsistencies with the Push method. It shows that the inconsistency follows Push<Invalidation<TTL due to the same reason as in Figure 14(a). Compared to Figure 14(a), nodes in the lower layer of the multicast tree with the TTL method have higher inconsistencies, since higher-layer nodes are closer to the provider and expected to receive the update earlier. For example, an update can reach nodes in layer 1 with a longest delay as TTL, but for nodes in layer 2, the longest delay will be $2 * TTL$. In general, a node in layer m has around $m - 1$ times the expected inconsistency compared to a node in layer 1.

Figure 15(b) shows the average of all inconsistencies of each end user with different update methods in the multicast-tree infrastructure, in which all servers are sorted by their inconsistencies with the Push method. We see that the inconsistencies of end-users for TTL increase compared to those in the unicast tree because of the increased inconsistencies

1045-9219 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2479222, IEEE Transactions on Parallel and Distributed Systems

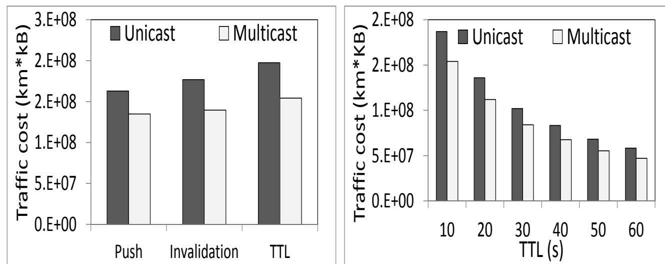


Fig. 16: Consistency maintenance cost.

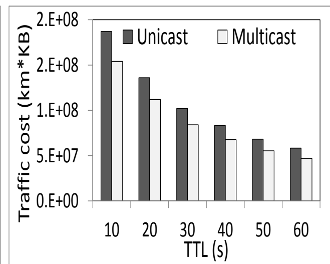
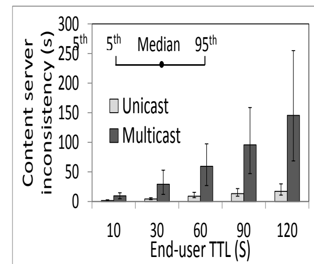
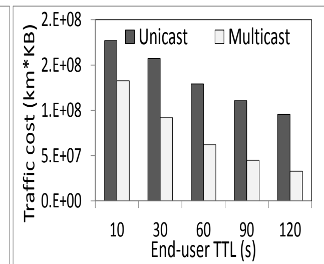


Fig. 17: Consistency maintenance cost vs. TTL of content servers.

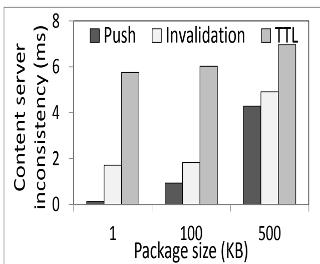


(a) Inconsistency

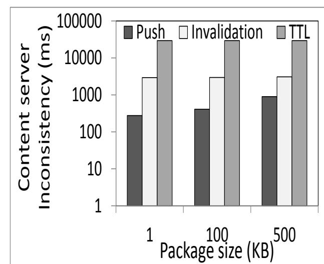


(b) Consistency maintenance cost

Fig. 18: Performance with varying end-user TTL in Invalidation.

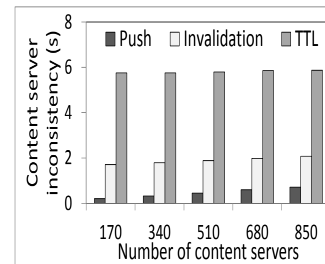


(a) Inconsistency in Unicast

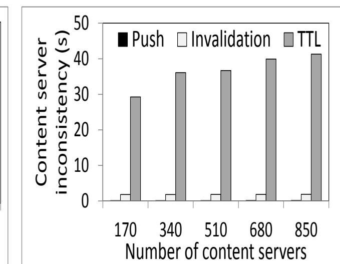


(b) Inconsistency in Multicast

Fig. 19: Scalability vs. update package size.



(a) Inconsistency in Unicast



(b) Inconsistency in Multicast

Fig. 20: Scalability vs. network size.

in the servers in the multicast tree. The other two methods in the multicast tree have the same performance as unicast tree.

4.3 Efficiency of Consistency Maintenance

As in [41], we measure the traffic cost as $km * KB$ for all packets for consistency maintenance. Figure 16 shows the total traffic cost of all update methods in both the unicast-tree and the multicast-tree. It shows that multicast can save at least $2.8 * 10^7 km * KB$ in traffic cost over unicast for all methods. This is because multicast trees are proximity aware, so updates are transmitted between proximity close nodes with short latency. In unicast, the provider needs to communicate with all servers distributed worldwide. The figure also shows that in both unicast and multicast, the traffic cost follows Push$<$Invalidation$<$TTL. In the trace, the update frequency is low. Thus, TTL wastes traffic in probing unchanged content. Invalidation has additional notification and polling packets compared to Push. As a result, Push generates lower traffic costs than Invalidation and TTL.

As shown in Figure 17, the overhead of consistency maintenance decreases as the time-to-live in the TTL method increases in both unicast and multicast. There are two reasons for this decrease: (a) the traffic overhead querying messages can be saved due to the larger time interval between queries; (b) larger time-to-live has a higher probability to skip an update. Since increasing the TTL has the same effect while decreasing the update frequency, it indicates that with frequent updates, TTL can be used for applications with weak consistency requirement to save consistency maintenance cost.

4.4 Inconsistency with Varying End-User TTL

Figure 18 shows the performance changes when TTL of cached content of end-users is varied from 10s to 120s. This TTL is only related to the Invalidation update methods. Thus we only measure the performance of Invalidation in both unicast and multicast. Figure 18(a) shows the 5%, median and 95% of average inconsistency of all servers for each TTL. It shows that both in unicast and multicast, the inconsistency increases as the TTL increases. That is because with longer end-user TTL, the expected time interval between the first content request and the invalidation increases due to the same reason as Figure 14(a), since the time-to-live is inverse proportional to the visit frequency. As shown in Figure 18(b), the overhead of consistency maintenance in traffic cost decreases as time-to-live increases in both unicast and multicast. This is because with larger end user TTLs, there may not be visits during two updates. Thus, it saves traffic costs by eliminating unused updates.

4.5 Infrastructure Scalability

Figure 19(a) shows the average content server inconsistency of different update methods in unicast tree. As the update packet size increases, the inconsistency increases due to higher traffic load and delay. The inconsistency increase rate as packet size increases follows Push$>$Invalidation$>$TTL. Push has the fastest inconsistency increase, because the provider needs to transmit updates to all servers upon updating. Larger update packages could cause congestion in the provider's uplink, which greatly increases the traffic latency of update packages. In Invalidation, the provider only sends a notification package to servers; thus, it is less likely to be overloaded. TTL generates the smallest increase rate because content servers' update requests are scattered in [0, TTL] after the updating time. Thus, the provider is much less likely to be overloaded. Figure 19(b) shows the average content server inconsistency of different update methods in multicast tree. It shows the same trend as in Figure 19(a) due to the same reasons. Comparing the two figures, we notice that the inconsistency increment of each method in multicast is smaller than that in unicast. The packet queuing delay is proportional to the package size and the number of children in the tree for pushing/responding to updates. Since a node has two children in multicast but has 170 children in unicast, the inconsistency in unicast increases much more rapidly than in multicast as packet size increases.

Figure 20(a) shows the average server inconsistency of different update methods in unicast tree for different network sizes. The inconsistency increases as the number of servers grows because more servers generate higher network load to the provider for transmitting updates. As the number of servers grows, the inconsistency increase rate follows Push$>$Invalidation$>$TTL due to similar reasons as in Figure 19(a). The content server inconsistency is stable in the TTL-based method, but it increases in Push and Invalidation as the number of content servers increases.

1045-9219 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2479222, IEEE Transactions on Parallel and Distributed Systems

The result indicates the high scalability of the TTL-based method when the network load is high.

Figure 20(b) shows the average server inconsistency of different update methods in multicast tree, where inconsistency in TTL is less than 0.14s. It is intriguing to see that TTL has the fastest increment; this occurs because larger network sizes increase the depth of the multicast tree, and the inconsistency is proportional to the depth of tree with an amplification factor in [0,TTL].

4.6 Summary of Trace-Driven Experimental Results

We summarize our experimental results below, which can provide guidance for selecting or designing a CDN update approach.

- Push provides better consistency on content servers than other methods in a small-scale network. However, its performance deteriorates rapidly in a large-scale network with heavy traffic burden.
- From the end user perspective, Invalidation can supply similar consistency guarantees as Push. It can also reduce traffic costs with infrequent visits from end users on frequently updated contents. However, for frequently updated contents, it introduces heavy network burden by transmitting the additional invalidation notifications.
- The TTL-based method can supply a weak consistency with inconsistency no larger than its TTL. It should have better scalability than the other two methods by releasing update transmission load of the content provider. However, it may waste unnecessary traffic costs on contents with infrequent updates.
- The proximity-aware multicast tree infrastructure can save more traffic costs and support better scalability than the unicast-tree infrastructure. However, it introduces much more inconsistency into the TTL-based method.

Given the varying performance of update methods, application developers can choose an update method based on their needs. For example, applications that require high consistency such as stock, e-commerce and live game webpages can use Push and unicast-tree infrastructure, while some applications that can tolerate small periods of inconsistency but need to avoid heavy overhead can use Invalidation or TTL-based methods depending on their degree of tolerance. For further network traffic reduction, the proximity-aware multicast tree infrastructure can be used. According to measurements, no single update method or an infrastructure supports both scalability and consistency in all scenarios. However, a combination of different methods with different infrastructures could work. New APIs may be needed to probe visit and update frequency of live contents, with which we can infer the changes of the scale of interested users. Additionally, considering customized requirements such as consistency, a self-adapting strategy could switch between update methods and infrastructures to find an optimal combination.

5 A Self-Adaptive Update Method On a Hybrid Infrastructure

From the conclusion of our experimental results, there is no single update method or a single infrastructure that can achieve both high scalability and high consistency simultaneously in all scenarios. Different update methods and infrastructures are suitable for different scenarios. We can use a self-adapting strategy that switches between the update methods and infrastructures to adaptively meet the specific features of a scenario.

Based on this guidance, in this section we propose a hybrid and self-adaptive update system that builds a hybrid update infrastructure and adaptively uses different update methods on the infrastructure. As a showcase, we consider the complex scenarios in our self-crawled trace on the major CDN. In the trace, the live game statistics have frequent updates during some time (during the match), and maintain silence for a long time (during the breaks). This update pattern has been found in other web applications, such as online social networks. In an online social network, the user data tends to be accessed heavily immediately after a post's creation, and the following comments for the post update the user data frequently, and then are rarely accessed without any updates [42], [43]. In this pattern, there are frequent updates due to a big event and subsequent update silence. Thus, for this complex scenario, the current update method using TTL based on unicast is not appropriate.

In the following, we first introduce our proposed self-adaptive update method that adaptively switches between the TTL and the Invalidation methods. We then introduce our proposed hybrid update infrastructure that uses both unicast and multicast. Finally, we demonstrate the outperformance of our hybrid and self-adaptive system compared to single update methods in achieving both high consistency and high scalability.

5.1 Self-Adaptive Update Method

In this section, we assume that inconsistency between the content provider and content servers is tolerable. Considering the much more frequent updates during the match, TTL is more appropriate than the other two methods, since it is scalable and can save the number of messages by aggregating the update messages within each TTL together. However, during the breaks, TTL is less appropriate than other update methods, since it generates unnecessary traffic. Therefore, in order to reduce the network load, we can use a self-adaptive update method that adaptively uses TTL or another update method when the update frequency becomes high or low.

Algorithm 1: Algorithm for the self-adaptive update.

```
1  Procedure Main()
2      TTL_based_update () ;
3  Procedure TTL_based_update()
4      do
5          Sleep for TTL;
6          Poll the update;
7      while ∃ an update;
8      Invalidation_based_update () ;
9  Procedure Invalidation_based_update()
10     Wait (an invalidation);
11     Wait (a visit);
12     Poll update and Notify switch from Invalidation to TTL;
13     TTL_based_update () ;
```

As a showcase of the self-adaptive update methods, we propose a simple method combining TTL with Invalidation. Compared to Push, the Invalidation may save more network load at the initial after an update silence, since there may not be a visit on a content server immediately after this update, the content provider can aggregate several updates together at the beginning of each match together until the first visit and send to the content server. Additionally, the first visits on different CDN serves may be different. If we switch to TTL after the first polling under Invalidation, different content servers may have different time to poll the contents periodically. The poll time diversity can avoid the congestions of the content provider. Accordingly, if we switch to TTL after the first push, the polling time of content servers are almost the same, which causes the Incast problem. In the Incast problem, a large number of data queries towards the content provider within a short time can easily overload it.

Our self-adaptive update method combines TTL and Invalidation. As shown in Algorithm 1, at the beginning, each content server first uses TTL to poll the update (Line 2). If there is

1045-9219 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2479222, IEEE Transactions on Parallel and Distributed Systems

no update during a TTL, the content server switches its update method to Invalidation and notifies the content provider (Line 8); otherwise, the content server polls the update after each TTL (Lines 4-7). The content provider maintains two types of update methods: the passive method based on TTL and the active method based on Invalidation. For the active method, the content provider needs to record all content servers using Invalidation. After a content server switches to the Invalidation method, it will not switch back to TTL until it first polls the update from the content provider. The first poll (Line 12) is generated after it receives the invalidation notification from the content provider, and there is a content query from client after this notification (Lines 10-11). Meanwhile, it needs to notify the content provider for the switching (Line 12), and then switches back to TTL (Line 13). The switching between TTL and Invalidation saves the number of update polling messages using the TTL while silence, and saves the frequent invalidation notification messages and aggregates the number of updates during a TTL together to reduce the number of update messages.

As indicated previously, the adaptive TTL methods [6], [8], [9], [18], [22], [24] may reduce traffic costs as well as support stronger consistency. Adaptive TTL requires that the update rate is predictable; otherwise, it is not effective in achieving consistency or scalability. However, the modification behavior of content is not necessarily regular, making consistency guarantees difficult. For example, a large TTL will be reduced when an update occurs much earlier than expected. If all subsequent updates occur at much longer intervals, periodic polling will occur unnecessarily. Our self-adaptive update method can avoid this problem. When there is no update in a poll, the content server will use the Invalidation update method. Then, it will not switch back to TTL until it first polls the update from the content provider after it receives invalidation message. Therefore, our method can be more adaptive to the update rate change.

5.2 Hybrid Update Infrastructure

Since some live matches are watched worldwide, we can assume that live statistics are also required by worldwide clients. Therefore, the contents will be distributed to the CDN's worldwide surrogate servers near the network edge. Using the current TTL method under unicast infrastructure, there are content servers far away polling the contents from the content provider. Plenty of updates to the content servers far away from the content provider will cause a heavy network load to the passed-through ISPs. In order to save network load, we can propagate the update to several supernodes, and let the supernodes to be responsible for the update polling of content servers nearby. Therefore, we need to group content servers together into clusters according to their geographical locations or their real cluster affiliation in the CDN. The CDN operator can know every server's location and affiliations. We first group users according to their affiliations, and then we group users according to their geographical locations based on [39]. In [39], the Hilbert curve [44] is used to convert two dimensions (longitude and latitude) to real numbers, named Hilbert numbers. Since physical close nodes will have similar Hilbert numbers, they group content servers based on the Hilbert number. For each group, there is a supernode, which is responsible for responding to update polling from servers in the same cluster. In order to save the network load of the supernode, all other servers adopt the self-adaptive update method.

To propagate updates to the supernodes from the content provider, the TTL update method may not be appropriate. Since the super nodes and other content servers form a multicast tree using TTL. As shown in Figure 15(a), the servers at the lower layer have much larger inconsistency, leading to twice the inconsistency. Therefore, we choose Push to actively propagate the updates to

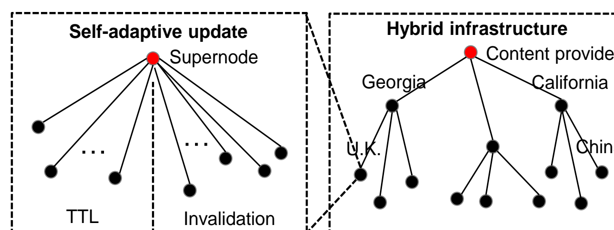


Fig. 21: The hybrid and self-adaptive update method.

the supernodes in time, without enlarging the inconsistency much more. However, the Push method under unicast does not scale well as shown in Figure 14(a). Therefore, we switch the infrastructure to multicast tree, with the content provider as the root. In order to save network load, we build a proximity aware k-ary multicast tree. Newly-joined supernodes or supernodes having lost parents choose the nearest supernode that has fewer than k children as its parent in the tree.

In summary, we build our hybrid infrastructure using both unicast and multicast, and combine all update methods including Push, Invalidation and TTL in order to propagate the updates in a network load efficient and scalable manner as shown in Figure 21. The content provider sends the updates to the supernodes in each cluster using Push. Inside each cluster, the content servers depend on the self-adaptive methods to get the updates. In this way, the content provider does not need to maintain two update methods, which simplify the CDN customer's functionality. The supernode instead plays the functionality as the content provider in Section 5.1.

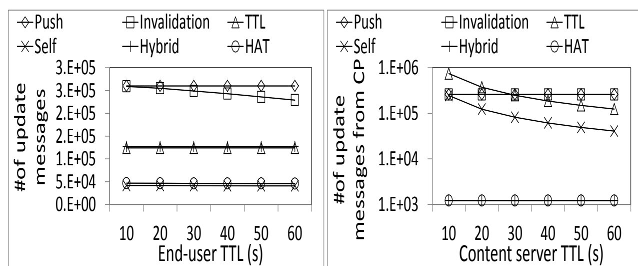
5.3 Performance Measurement

In this section, we verify the effectiveness of our proposed simple hybrid and self-adaptive method in saving the network load under the scenario in the trace. We use the same settings in Section 4. We clustered the PlanetLab nodes into 20 clusters according to their locality. In order to test at a large scale, each PlanetLab node simulates 5 content servers, and each content server has five observers. The supernode is randomly chosen from the node in the cluster, and all supernodes form a 4-ary tree. Unless specified, the TTL of the content servers is set to 60s, and the TTL for the observer is set to 10s. We use HAT to represent our Hybrid and self-AdapTive method. We also compare our method with Push, Invalidation and TTL under unicast, the self-adaptive update method under unicast (denoted by Self), and our Hybrid update infrastructure using TTL update method inside each cluster (denoted by Hybrid).

The size of an update message is usually much larger than the size of other messages, termed light messages and including update polling messages, invalidation messages and structure maintenance messages. Therefore, we use the number of update messages to indicate the network load including the polling responses and update messages. Figure 22(a) measures the total number of update messages to the content servers versus the TTL of end users. It shows that the number of update messages follows Push>Invalidation>Hybrid≈TTL>HAT>Self. Since the update rate is much larger than the content server's TTL, Push has many more messages than the methods using TTL, including Hybrid, TTL, HAT and Self. Also, Invalidation can skip some update messages to a content server, if between the update and its successors there are no visits on this content server. Therefore, Push has the largest number of update messages. Since there are five end-users on each content server, the visit frequency on a content server is high enough to have a visit between each update and its succeed update with high probability. Therefore, Invalidation skips very few messages compared to TTL, and

1045-9219 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2479222, IEEE Transactions on Parallel and Distributed Systems



(a) Number of update responses (b) Number of update responses from the content provider

Fig. 22: Performance of update message saving.

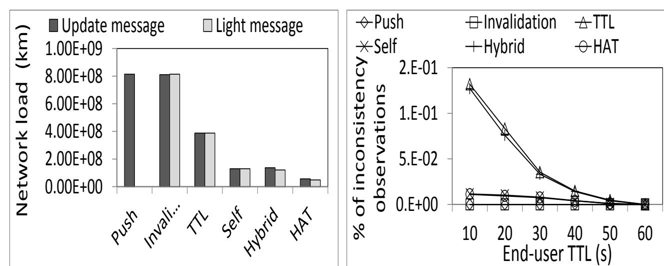


Fig. 23: Consistency maintenance network load (km).

Fig. 24: % of inconsistency observations.

generates many more messages than the TTL based methods. The figure also shows that the number of update messages in Invalidation decreases while the end-user TTL increases. Also, the visit frequency decreases while the end-user TTL increases, leading to a lower probability that there is a visit after each update. TTL and Hybrid generate similar number of messages, since the majority of servers in both methods use TTL. The Self and HAT methods use the self-adaptive update method, switching between TTL and Invalidation to save update messages in both frequent update and rare update scenarios. Therefore, they generate fewer messages than TTL and Hybrid. The Self method generates fewer messages than HAT, because HAT uses Push to send updates to supernodes, leading to more update messages. However, as Self does not consider proximity, it generates much more network load than HAT. The figure indicates that HAT generates nearly the smallest number of messages in order to save network load.

We further measure the network load of the content provider in order to show HAT's effectiveness in saving the network load of content provider to improve scalability. Figure 22(b) shows the number of update messages from the content provider versus the TTL of content provider. In Push, Invalidation, TTL and Self, all updates are from the content provider directly. Thus, the figure shows the same orders among these methods as shown in Figure 22(a) due to the same reason, when the content server TTL equals to 60s. Since the Hybrid and HAT methods both use the multicast infrastructure for content provider to disseminate the updates, only the four content servers at the second layer in the tree receive updates from the content provider directly. There, they generate the lightest network load for the content provider. It also shows that TTL and Self generate more update messages while the TTL decreases, that is because shorter TTL leads to larger content polling frequency and get more update messages. The figure indicates that HAT generates the smallest network load for the content provider.

We then measure the network load by considering the total transmission distance of update messages in kilometers. Figure 23 shows the network load of all systems for update messages and light messages, respectively. It shows a similar order among all methods as in Figure 22(a), since more messages lead to larger network load. Different from Figure 22(a), it shows that Hybrid generates much smaller network load for update messages than TTL by considering locality. Therefore, Hybrid generates a similar update network load as Self, despite generating many more messages. It also shows that HAT generates the lightest update network load since it generates very few update messages with locality awareness. Also, all Invalidation or TTL based methods generate almost the same number of light messages as of the update messages due to the update polling. Considering all network load together, HAT still generates the lightest network load. The figure indicates that HAT successfully saves more network load than other methods by reducing the number of update messages and propagating updates with locality awareness.

We then measure the user observed inconsistency as shown in Figure 24. The percentage of inconsistency observations is calculated by the number of observations receiving content older than current content from all users divided by the total number of observations. We evaluate the performance under a scenario in which a user switches to another content server for every successive visit. It shows that percentage of inconsistency observations follows TTL≈Hybrid>HAT>Self>Push≈Invalidation≈0. Since Push and Invalidation do introduce a very small inconsistency to the content servers, the users hardly observe any inconsistency. Therefore, they generate fewer inconsistency observations than other methods based on TTL. We can observe that all TTL methods generate fewer inconsistency observations while the end-user TTL increases. Because with a longer end-user TTL approximate to the content server's TTL, there is a higher probability that the server switched to will have the latest content. We can also say that the self-adaptive update method has fewer inconsistency observations than the other methods based directly on TTL. This is because after each update silence, the first visit on each content server is expected to arrive in a short time than the end-user TTL. Therefore, all content servers start the TTL update method within a shorter period, leading to less inconsistency between each other. Thus, users have lower probability to observe an inconsistency. The figure indicates that HAT improves the user observed inconsistency compared to other methods based on TTL.

6 Conclusions

In this paper, we analyzed our crawled trace data of cached sports game content on thousands of servers in a major CDN. From our analysis, we noted that the inconsistency problem does exist in the CDN. From the analysis, we not only comprehensively evaluated the inconsistency of dynamic contents among the CDN's servers, but also broke down the reasons for inconsistency among end-users. We then verified that the CDN adopts unicast as the infrastructure, which may introduce too much network load. Finally we evaluated the performance in consistency and overhead for different infrastructures with different update methods and itemized the advantages and disadvantages. Based on the evaluation, we further proposed our hybrid and self-adaptive method to save network load under the scenario in the trace, and use trace-drive experiments to validate its effectiveness. In this paper, we aim to give guidance of appropriate selections of consistency maintenance infrastructures and methods when building a CDN or choosing a CDN service. In our future work, we will study a more generic hybrid and self-adaptive consistency maintenance method that can change the update method and infrastructure by considering more factors, such as varying visit frequencies and consistency requirements from customers.

References

[1] D. Rayburn. CDN Market Getting Crowded: Now Tracking 28 Providers In The Industry. *Business of Online Video Blog*, 2007.

[2] Akamai. http://www.akamai.com/.

1045-9219 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2479222, IEEE Transactions on Parallel and Distributed Systems

[3] M. Zhao, P. Aditya, Y. Lin, A. Harberlen, P. Druschel, W. Wishon, and B. Maggs. A First Look at a Commercial Hybrid Content Delivery System. http://research.microsoft.com/apps/video/default.aspx?id=154911.
[4] C. Huang, A. Wang, J. Li, and K. W. Ross. Measuring and Evaluating Large-Scale CDNs. In *Proc. of IMC*, 2008.
[5] G. Pallis and A. Vakali. Insight and Perspectives for Content Delivery Networks. *Communications of the ACM*, 2006.
[6] Z. Fei. A novel approach to managing consistency in content distribution networks. In *Proc. of WCW*, 2001.
[7] P. Cao and C. Liu. Maintaining Strong Cache Consistency in the World-Wide Web. In *Proc. of the Seventeenth International Conference on Distributed Computing Systems*, 1997.
[8] X. Tang, J. Xu, and W. Lee. Analysis of TTL-Based Consistency in Unstructured Peer-to-Peer Networks. *IEEE TPDS*, 2008.
[9] Z. Wang, S. K. Das, H. Che, and M. Kumar. A Scalable Asynchronous Cache Consistency Scheme (SACCS) for Mobile Environments. *IEEE TPDS*, 2004.
[10] J. Lan, X. Liu, P. Shenoy, and K. Ramamritham. Consistency Maintenance in Peer-to-Peer File Sharing Networks. In *Proc. of WIAPP*, 2003.
[11] A. Datta, M. Hauswirth, and K. Aberer. Updates in Highly Unreliable, Replicated Peer-to-Peer Systems. In *Proc. of ICDCS*, 2003.
[12] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *Proc. of the International Workshop on Design Issues in Anonymity and Unobservability*, 2001.
[13] A. Ninan, P. KulkarniI, P. Shenoy, K. Ramamritham, and R. Tewari. Cooperative Leases: Scalable Consistency Maintenance in Content Distribution Networks. In *Proc. of WWW*, 2002.
[14] M. Roussopoulos and M. Baker. CUP: Controlled Update Propagation in Peer to Peer Networks. In *Proc. of USENIX*, 2003.
[15] L. Yin and G. Cao. Ynamic-Tree Based Update Propagation in Peer-to-Peer Networks. In *Proc. of ICDE*, 2005.
[16] X. Chen, S. Ren, H. Wang, and X. Zhang. SCOPE: Scalable Consistency Maintenance in Structured P2P Systems. In *Proc. of INFOCOM*, 2005.
[17] Z. Li, G. Xie, and Z. Li. Locality-Aware Consistency Maintenance for Heterogeneous P2P Systems. In *Proc. of IPDPS*, 2007.
[18] H. Shen. GeWave: Geographically-Aware Wave for File Consistency Maintenance in P2P Systems. In *Proc. of ICPP*, 2008.
[19] X. Tang, H. Chi, and S. T. Chanson. Optimal Replica Placement under TTL-Based Consistency. *IEEE TPDS*, 2007.
[20] Z. Feng, M. Xu, Y. Wang, and Q. Li. An Architecture for Cache Consistency Support in Information Centric Networking. In *Proc. of GLOBECOM*, 2013.
[21] V. Cate. Alex: A Global File System. In *Proc. of the USENIX File System Workshop*, 1992.
[22] H. Shen. IRM: Integrated File Replication and Consistency Maintenance in P2P Systems. *TPDS*, 2010.
[23] H. Shen and G. Liu. A Geographically Aware Poll-Based Distributed File Consistency Maintenance Method for P2P Systems. *IEEE Trans. Parallel Distrib. Syst.*, 2013.
[24] C. Chen, S.Matsumoto, and A. Perrig. ECO-DNS: Expected Consistency Optimization for DNS. In *Proc. of ICDCS*, 2015.
[25] N. Diegues and P. Romano. STI-BT: A Scalable Transactional Index. In *Proc. of ICDCS*, 2014.
[26] P. Shang, S. Sehrish, and J. Wang. TRAID: Exploiting Temporal Redundancy and Spatial Redundancy to Boost Transaction Processing Systems Performance. *IEEE TC*, 2012.
[27] E. Nygren, R. K. Sitaraman, and J. Sun. The Akamai Network: A Platform for High-Performance Internet Applications. In *Proc. of SIGOPS*, 2010.
[28] A. Thomson and D. J. Abadi. CalvinFS: Consistent WAN Replication and Scalable Metadata Management for Distributed File Systems. In *Proc. of FAST*, 2015.
[29] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *Proc. of NSDI*, 2013.
[30] H. Abu-Libdeh, R. Renesse, and Y. Vigfusson. Leveraging Sharding in the Design of Scalable Replication Protocols. In *Proc. of SoCC*, 2013.
[31] X. Tang and S. Zhou. Update Scheduling for Improving Consistency in Distributed Virtual Environments. *TPDS*, 2010.
[32] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues. When Scalability Meets Consistency: Genuine Multiversion Update-Serializable Partial Data Replication. In *Proc. of ICDCS*, 2012.
[33] J. Xiong, Y. Hu, G. Li, R. Tang, and Z. Fan. Metadata Distribution and Consistency Techniques for Large-Scale Cluster File Systems. *TPDS*, 2011.
[34] T. Distler and R. Kapitza. Optimal Algorithms and Approximation Algorithms for Replica Placement with Distance Constraints in Tree Networks. In *Proc. of EuroSys*, 2011.
[35] X. A. Dimitropoulos, P. Hurley, A. Kind, and M. Ph. Stoecklin. On the 95-percentile billing method. In *PAM*, 2009.
[36] Best Practices when Developing with Akamai. http://http://seabourneinc.com/2011/04/28/best-practices-when-developing-with-akamai/.
[37] Iplocation. http://www.iplocation.net/.
[38] V. Valancius, C. Lumezanu, N. Feamster, R. Johari, and V. V. Vazirani. How Many Tiers?: Pricing in the Internet Transit Market. In *Proc. of SIGCOMM*, 2011.
[39] H. Shen and G. Liu. A Lightweight and Cooperative Multifactor Considered File Replication Method in Structured P2P Systems. *IEEE Trans. Computer*, 2013.
[40] PlanetLab. http://www.planet-lab.org/.
[41] M. P. Wittie, V. Pejovic, L. B. Deek, K. C. Almeroth, and Y. B. Zhao. Exploiting Locality of Interest in Online Social Networks. In *Proc. of ACM CoNEXT*, 2010.
[42] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebooks Distributed Data Store for the Social Graph. In *Proc. of ATC*, 2013.
[43] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a Needle in Haystack: Facebook's Photo Storage. In *Proc. of OSDI*, 2010.
[44] Z. Xu, M. Mahalingam, and M. Karlsson. Turning heterogeneity into an advantage in overlay routing. In *Proc. of INFOCOM*, 2003.

Guoxin Liu received the BS degree in BeiHang University 2006, and the MS degree in Institute of Software, Chinese Academy of Sciences 2009. He is currently a Ph.D. student in the Department of Electrical and Computer Engineering of Clemson University. His research interests include distributed networks, with an emphasis on Peer-to-Peer, data center and online social networks. He is a student member of IEEE.

Haiying Shen received the BS degree in Computer Science and Engineering from Tongji University, China in 2000, and the MS and Ph.D. degrees in Computer Engineering from Wayne State University in 2004 and 2006, respectively. She is currently an Associate Professor in the ECE Department at Clemson University. Her research interests include distributed computer systems and computer networks with an emphasis on P2P and content delivery networks, mobile computing, wireless sensor networks, and grid and cloud computing. She was the Program Co-Chair for a number of international conferences and member of the Program Committees of many leading conferences. She is a Microsoft Faculty Fellow of 2010, a senior member of the IEEE and a member of the ACM.

Harrison Chandler received his BS degree in Computer Engineering from Clemson University in 2012. He is currently a Ph.D. student in the Department of Electrical Engineering and Computer Science at University of Michigan. His research interests include embedded and distributed systems.

Jin Li received the PhD (with honor) degree from Tsinghua University in 1994. After brief stints at USC and Sharp Labs, he joined Microsoft Research in 1999, first as one of the founding members of Microsoft Research Asia, and then moved to Microsoft Research (Redmond, WA) in 2001. From 2000, he has served as an affiliated professor in Tsinghua University. He manages the Compression, Communication and Storage group. Blending theory and system, he excels at interdisciplinary research, and is dedicated to advance communication and information theory and apply it to practical system building. He is a research manager and principal researcher at Microsoft Research, Redmond, Washington. His invention has been integrated into many Microsoft products. Recently, he and his group members have made key contributions to Microsoft product line (e.g., RemoteFX for WAN in Windows 8, Primary Data Deduplication in Windows Server 2012, and Local Reconstruction Coding in Windows Azure Storage), that leads to commercial impact in the order of hundreds of millions of dollars. He was the recipient of Young Investigator Award from Visual Communication and Image Processing98 (VCIP) in 1998, ICME 2009 Best Paper Award, and USENIX ATC 2012 Best Paper Award. He is/was the associate editor/guest editor of the IEEE Transaction on Multimedia, Journal of Selected Area of Communication, Journal of Visual Communication and Image Representation, P2P networking and applications, and Journal of Communications. He was the general chair of PV2009, the lead program chair of ICME 2011, and the technical program chair of CCNC 2013. He is a fellow of the IEEE.

1045-9219 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.