Information and Media Technologies 7(2): 601–613 (2012)
reprinted from: IPSJ Transactions on Advanced Computing Systems 5(2): 10–22 (2012)
© Information Processing Society of Japan

**Regular Paper**

# MashCache: Taming Flash Crowds by Using Their Good Features

Hikaru Horie[1,a) Masato Asahara[2] Hiroshi Yamada[1] Kenji Kono[1]

Received: July 19, 2011, Accepted: December 30, 2011

***Abstract:*** This paper presents MashCache, a scalable client-side architecture for taming harsh effects of flash crowds. Like previous systems, MashCache extensively makes use of client resources to cache and share frequently accessed contents. To design MashCache, we carefully investigated the quantitative analysis of flash crowds in the wild and found out that flash crowds have three good features that can be used to simplify the overall design of MashCache. First, flash crowds start up very slowly. MashCache has enough time to prepare for the coming flash crowds. Second, flash crowds last for a short period. Since the accessed content is rarely updated in this short period, MashCache can employ a simple mechanism for cache consistency. Finally, the clients issue the same requests on the target web sites. This feature simplifies the cache management of MashCache because the cache explosion can be avoided. By using these good features of flash crowds, MashCache advances the state-of-the-art of P2P based caching systems for flash crowds; MashCache is a pure P2P system that combines 1) aggressive caching, 2) query-origin key, 3) two-phase delta consistency, and 4) carefully designed cache meta data. MashCache has another advantage. Since it works completely on client-side, MashCache can tolerate flash crowds even if the external services the mashup depends on are not equipped with flash-crowd-resistant mechanisms. In the experiments with up to 2,500 emulated clients, a prototype of MashCache reduces the number of requests to the original web servers by 98.2% with moderate overheads.

***Keywords:*** flash crowds, content delivery, peer-to-peer, mashups

## 1. Introduction

Web servers are not so stable and often fall down when "flash crowds" occur. In flash crowds, web servers become unavailable because a huge number of clients suddenly access the contents managed by the web servers. Flash crowds are quite difficult to deal with because it occurs unpredictably and the scale of flash crowds is unknown. Considering the unpredictability of flash crowds, many previous systems use clients' resources to defend against flash crowds. This is because the total resources of the clients naturally increase, when flash crowd occurs, in proportion to the number of clients that access the contents of interest. If we can make use of the clients' resources, we can use as many resources as needed to mitigate flash crowds; we do not have to predict the scale of and prepare resources for flash crowds.

This paper describes the design and implementation of *MashCache*, a scalable client-side architecture to mitigate flash crowds. Like many previous systems, MashCache constructs a distributed hash table (DHT) to cache and share frequently accessed contents among the clients. The design of MashCache is carefully derived from the quantitative analysis of flash crowds. Flash crowds have three good features that can be leveraged to simplify the overall design and mechanism of MashCache. First, flash crowds start up very slowly. For example, a flash crowd reported in Jung et al. [1] takes 4,100 seconds to reach its peak access rate. Therefore, we can take a reactive approach to tame flash crowds. Second, flash crowds last for a short period. For example, a flash crowd lasts for 100 minutes [1]. Lastly, only a small portion of the contents is accessed in flash crowds. For example, less than 10% of most popular documents account for more than 90% of requests [1], [2]. These two good features of flash crowds combine to simplify the cache management in MashCache, and allows us to take a radical approach to taming flash crowds. Since the contents accessed frequently during a flash crowd are rarely updated in this short period, we can employ simple and very weak semantics of cache consistency. Since only a small portion of the contents is frequently accessed, the amount of the contents cached in MashCache is not so large. MashCache can use clients' resources extensively because we do not have to worry about cache explosion. The cached contents will be revoked soon because flash crowds do not last for a long time.

Based on these observations, MashCache employs a cache management policy called *aggressive caching*. Aggressive caching is aggressive in two points. First, it creates a cache of any content accessed by clients, and replicates it whenever possible. By doing this, the target contents of flash crowds are always cached all over the DHT. Second, the cache expires very quickly to be revoked from the DHT. By aggressively creating and revoking the cache, only the contents accessed heavily continue to be cached in MashCache.

To simplify the cache consistency management, MashCache employs *two-phase delta consistency*, a minor extension to the

1 Keio University, Yokohama, Kanagawa 223–8522, Japan
2 NEC Service Platforms Research Laboratories, Kawasaki, Kanagawa 211–8666, Japan
a) hikaru.horie@sslab.ics.keio.ac.jp

Information and Media Technologies 7 (2) : 601–613 (2012)
reprinted from: IPSJ Transactions on Advanced Computing Systems 5 (2) : 10–22 (2012)
© Information Processing Society of Japan

delta consistency model. Two-phase delta consistency prevents original web servers from being overloaded on cache expiration. In MashCache, the cache is created almost at the same time when a flash crowd starts. So, many clients may access the original web server to confirm whether the cache is up-to-date or not. Two-phase delta consistency regulates the access to original servers to avoid an avalanche of confirmation messages.

By making use of the good features of flash crowds, MashCache also simplifies DHT key management. As mentioned above, a small percentage of the contents account for a large fraction of the requests. So, we can use a HTTP query (e.g., URL, parameters of GET/POST methods, cookies) as a key to access DHT. No index server is required to map the query to DHT keys. This differs from other systems such as BitTorrent [3], Avalanche [4], and Flashback [5]. Since there is no index server in MashCache, MashCache is hotspot-free and scalable to many client accesses. However, the use of queries as DHT keys poses another problem of privacy leak since URLs often contain privacy information such as cookies. Instead of using queries directly as DHT keys, MashCache uses hash values calculated from the queries.

Our design of MashCache brings another benefit. MashCache is suitable for handling *mashups*. Mashups are current trends in web, which combine two or more external services to create a rich and complex service. Unfortunately, mashup services make it harder for web services to be resistant to flash crowds. Since a mashup contains external services managed by someone else, the mashup provider cannot provision the external services with sufficient resources even though the flash crowd could be predicted. Even if a mashup itself does not suffer from flash crowds, it becomes unstable if one of the external services suffers from flash crowds; the bad effect of flash crowds is propagated easily to other services in mashups. Since MashCache works completely on client-side, mashups can be made resistant to flash crowds; no external services have to be equipped with flash-crowd-resistant mechanisms.

We implemented a prototype of MashCache using Overlay-Weaver 0.9.9 [6], a toolkit for developing overlay networks. To confirm that MashCache reduces the load on web servers, we conduct some experiments with up to 2,500 emulated clients. The experimental results demonstrate that MashCache reduces the number of requests on original web servers by 98.2% with moderate overhead.

The rest of the paper is organized as follows. Section 2 describes our analysis of flash crowds. In Section 3 we describe the design of MashCache. Section 4 describes the implementation of MashCache. In Section 5, we evaluate MashCache through several experiments. In Section 6, we discuss overhead and security of MashCache. Section 7 describes the work related to MashCache. Then we conclude the paper in Section 8.

## 2. Analysis of Flash Crowds

In this section, we show a quantitative analysis of flash crowds. First, we show the difficulties in managing flash crowds, using real examples of flash crowds that occurred in the past. Second, we describe our findings that can be leveraged to tame flash crowds. Our design of MashCache is based on this detailed analysis.

### 2.1 Bad Features

It is almost impossible to predict when flash crowds occur and how large they are because they are caused by many phenomena (e.g., accidents, natural disasters, rumors or *Slashdot effects* [7], [8]). This unpredictability is essential in considering solutions to flash crowds.

The unpredictability of flash crowds makes proactive approaches less attractive. In proactive approaches, we have to estimate the maximum load and prepare resources enough to handle flash crowds. Even if the maximum load could be estimated, the resources prepared for flash crowds are surplus and dissipate power and costs in normal operations. In flash crowds, access frequency increases by up to several hundred or thousand times. Jung et al. reported two real case of flash crowds using access logs of two real web sites; one is the play-along web site for a popular TV show and the other is Chilean election site hosted the continuously updated results of the 1999 presidential election in Chile [1]. In the first case, the frequency of requests increased from few requests per second to 6,719 requests per second within 4,100 seconds. In the second case, it increased from zero requests per second to about 300 requests per second within 40 seconds.

Although cloud computing platforms such as Amazon EC2 [9] and Google App Engine [10] provide elastic computing capacity, the cloud computing alone does not solve all the problems caused by flash crowds. We can not use resources of these services infinitely because the number of their host servers are limited and we have to keep expenditures within our budget. In addition, server-side solutions are difficult to use for mashups because we need to install the server-side mechanism on all the external servers the mashup consists of.

### 2.2 Good Features

#### 2.2.1 Flash Crowds Start Up Very Slowly

Flash crowds start up very slowly. Request rates reach the peaks in the order of seconds. Jung et al. investigated two real case of flash crowds using access logs of two real web sites; one is the play-along web site for a popular TV show and the other is Chilean election site hosted the continuously updated results of the 1999 presidential election in Chile [1]. In the first case, the frequency of requests increased from few requests per second to 6,719 requests per second within 4,100 seconds. In the second case, it increased from zero requests per second to about 300 requests per second within 40 seconds. Freedman also shows there was no domain which experienced 10-fold increase in 76% in less that 5 seconds [2]. This feature allows us to take a *reactive* approach to taming flash crowds. MashCache has enough time to prepare for cache of target contents of flash crowds. Therefore, we can deal with the problem caused by unpredictability of flash crowds if we can efficiently use client resources increase in flash crowds to reduce the load of web servers.

#### 2.2.2 Short Liveness

Flash crowds continue only for short periods. Jung et al. showed that the two flash crowds they observed continued

100 minutes and 282 minutes, respectively [1]. Freedman also showed two records of flash crowds because of Slashdot effects on two web sites using CoralCDN [2]. The former was on a site linked to a Slashdot article in May 2005. After rising, the flash crowds lasted less than three hours in duration and concluded abruptly. The latter was on a mirror site for a popular portal site (reddit.com) and a less popular portal site (moonbuggy.com) in August 2009. These two sites exposed to the flash crowds for half a day a few hours after flash crowds occurred on reddit.com. This feature allows us to avoid maintaining caches for long time. MashCache can immediately discard caches when the flash crowds calm down.

#### 2.2.3 Limited Targets of Requests

A large fraction of requests target a small fraction of the contents. Less than 10% of most popular documents account for more than 90% of requests by traces of the play-along site and the Chilean election site respectively [1]. Freedman [2] shows that, in CoralCDN, 0.01% of URLs account for 49.1% of requests and 10% of URLs account for 92.2% of requests in the flash crowds occurred in August 2009. This feature allows us to effectively reduce the load on the web servers with a small repertoire of caches. MashCache can avoid explosion of cache data even if MashCache tries to create and keep caches for all contents each client acquired.

## 3. Design

### 3.1 Design Issues

MashCache utilizes client resources to handle flash crowds for the following reasons. First, when flash crowds occur, the total resources of the clients naturally increase in proportion to the number of clients that access the contents of interest. Resources are not to be prepared for flash crowds in advance. Second, no server-side mechanism is needed to handle flash crowds. This indicates that the approach is applicable to services like mashups that consist of many external services. In designing MashCache, the following design challenges must be addressed.

- *Prediction-Free:* As described in Section 2, it is quite difficult to predict when flash crowds occur. Employing a mechanism that does not need the prediction is suitable for handling flash crowds.
- *Managing numerous client resources:* To efficiently use client resources, we need a scalable mechanism for managing a huge number of clients.
- *Load Balance:* To avoid hot spots in MashCache, the load from clients must be uniformly distributed among the machines in MashCache.
- *Consistency:* To keep the consistency of contents, a mechanism that efficiently propagates updates is needed. If the consistency is not kept properly, clients might continue to get older contents.

### 3.2 Solutions

To address these challenges, we make use of the good features of flash crowds described in Section 2. MashCache takes advantage of the state-of-the-art P2P mechanisms. MashCache is a pure P2P system that combines 1) aggressive caching, 2) query-origin key, 3) Two-phase Delta Consistency, and 4) carefully designed cache meta data. **Figure 1** shows a high level design of MashCache. Clients participate in an overlay network for sharing caches. To get a content, a client tries to get its cache from MashCache overlay. If there is not a cache in MashCache overlay, the client sends a request to the original server that provides the content and puts a copy of the content into MashCache. In these procedures, MashCache does not rely on any special servers to manage them.

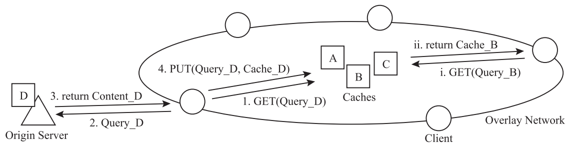

**Fig. 1** High level design of MashCache.

#### 3.2.1 Pure P2P Structure

MashCache takes a pure P2P structure to manage clients and caches. Using a P2P structure, MashCache can aggregate the resources of its participating clients. When the number of clients that access to a content increases, the total amount of the available resource also increases automatically in the P2P structure. Therefore, MashCache does not need to predict when flash crowds occur and how large they are. This P2P structure releases MashCache from any special servers for managing clients and caches. Our prototype of MashCache uses a DHT (more concretely, Chord) to manage clients and caches.

#### 3.2.2 Aggressive Caching

MashCache employs an aggressive caching policy, in which all the contents that the clients get from their origin servers are cached in MashCache. This policy releases us from the hard task to forecast which contents would get popular. At first glance, this aggressive caching has two drawbacks. First, the caching mechanism does not work in a timely fashion. When a severe flash crowd occurs, the accessed contents are not replicated quickly enough because the cache is created *after* the content is accessed by a client. Second, the cache may explode because all the contents are cached.

Fortunately, the good features of flash crowds advocate the use of aggressive caching. Since a flash crowd starts up slowly (Section 2.2.1), the caching mechanism has enough time to replicate the frequently accessed contents. When the number of the clients increase, the cache of the accessed contents are also replicated in MashCache nodes. The other features (short liveness and limited target of requests) combine to reduce the total amount of resource needed to maintain caches. Due to the short liveness of flash crowds, the cache can expire soon so that it can be revoked quickly when the flash crowd finishes. In addition, the total amount of cache is suppressed automatically since only a small portion of the contents is heavily accessed in flash crowds.

#### 3.2.3 Query Origin Key

MashCache uses HTTP queries (e.g., URL, parameters of GET/POST, Cookies) as keys to access the underlying DHT. As described in Section 2.2.3, a large fraction of requests target only a small portion of the contents during flash crowds. So, the use of HTTP queries as keys to access DHT is attractive. This design of MashCache also releases us from the use of an index server that binds keys and caches in the DHT. However, this design has a critical drawback. Since the HTTP query contains private information such as cookies, we cannot use them directly as keys to DHT. To hide the details of HTTP queries, MashCache introduces *Query Origin Key*. Query origin keys are generated by applying a hash function to HTTP queries. By doing this, we can hide the details of HTTP queries and use uniform, fixed-length keys to access the DHT.

Our prototype assumes that the hash function generates a completely unique value for each unique input. In fact, it is possible for the hash function to generate the same value even for different. The hash conflict causes MashCache to return wrong caches to clients. To avoid this problem, we can extend MashCache in two ways. The first is to encrypt caches by using original HTTP queries as encryption keys. In this way, each client encrypts a cache when the client uploads it to the overlay network. If another client gets the cache with the same hashed value of another HTTP query, the client cannot decrypt the cache because the client uses a different HTTP query as the encrypt key. Although this encryption technique is an effective way to avoid the hash conflict, we suffer from some overhead incurred by the encryption and decryption.

The second way is confirmation by using other hash functions. In this way, each client writes multiple hashed values generated by other hash functions into Cache Meta Data which can contain additional information about a corresponding cache. Other clients can confirm by using multiple hashed values and avoid to get not-desired-cache because it is quite rare case that all of them get corrupted. We describe more details of Cache Meta Data in next section.

### 3.2.4 Cache Meta Data

Since a query-origin key is assigned to each HTTP query, the client that manages a certain key becomes overloaded if the content correspoding to the key is heavily accessed during flash crowds. To avoid this problem, MashCache introduces *Cache Meta Data* (*CMD*) that contains additional information for cache. CMD enables us to manage caches more flexibly and solve many subtle problems.

As shown in **Fig. 2**, the query-origin key is mapped to CMD that manages the caches of the corresponding contents. CMD is composed of several keys that are used to access other data in the DHT. To distribute the access to a certain content, MashCache replicates the content and assign different keys to the replicas. Then, MashCache creates a CMD that contain those keys to the replicas and puts it in the DHT with the query-origin key. Using this CMD, the clients can access the replicas and thus, the load is distributed among the replicas.

CMD can be used to split a large-sized cache to chunks to distribute the load among several clients. If a large-sized cache is split into $n$ chunks, the corresponding CMD contains $n$ keys to access the chunks. The client accessing the content gets the CMD, collects all the chunks, and combine them. CMD can be also used to encrypt the raw data stored in the DHT.

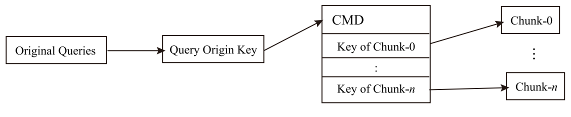

**Fig. 2** Pointing chunks through CMD.

### 3.2.5 Two-phase Delta Consistency

Two-phase Delta Consistency enables us to keep caches fresh and to reduce the load on web servers. **Figure 3** shows that two-phase delta consistency achieves frequent cache updates without a rush in the web servers under the many clients send requests. MashCache employs a simple cache management system that is based on time-to-live (TTL). In TTL-based cache management system, when the cache expires, the caching system asks the original servers to know if the cache is still up-to-date or not. In MashCache, the caches of a single content is created on many clients almost at the same time when a flash crowd occurs. If the naive TTL-based caching is used, the caches expire at the same time and many clients rush to the original server to check validity of the cache (illustrated as a series of black triangles in the period in Fig. 3). To avoid this problem, MashCache employs two-phase delta consistency, an extension to the delta consistency.

In the two-phase delta consistency, each client manages two expiration times (minor $\delta$ and major $\Delta$ illustrated in Fig. 3) to decide how to deal with each cache (CMD/chunk) managed by itself. Like a general TTL-based caching, the client invalidates the cache when the major expiration time $\Delta$ passes by. When the minor expiration time $\delta$ passes by, the client responsible to the cache deceives one of the accessing clients into getting a up-to-date content from the original web server. On the other hand, the other clients continue to access the caches maintained by the responsible client (illustrated as a series of white triangles between the time a cache expires and a new cache is deployed in Fig. 3). By doing this, only the deceived client gets the original contents from the web servers and deploys them as the new cache.

We define $\delta$ and $\Delta$ ($\delta < \Delta$) as intervals to set the minor expiration time and the major expiration time for each cache (details in Section 4.1). Each client sets new $\delta$ when it misses the minor expiration time and the major expiration time when the cache is created or updated.

## 4. Implementation

MashCache works completely on the client-side and we assume that it works as, for example, a browser plug-in or a local proxy daemon. **Figure 4** shows an architecture of MashCache. *Proxy* receives requests from a user application such as a web browser. *Core* controls the underlying five components; *DHT*, *messaging*, *downloader*, *data processor*, and *key generator*.

DHT, messaging, and downloader components interact with other MashCache clients via network. The DHT component manages all clients, CMD, and chunks. MashCache makes use of the DHT to locate CMD and chunks. Although MashCache can employ any DHT routing algorithms, the employed algorithm should be scalable since there is a huge number of clients under flash crowds. The messaging component manages messages between clients. The messages are sent to request and response for getting and putting CMD and chunks. The downloader component fetches the contents from the original servers when they are not cached yet.

The data processor plays a role of processing contents. It gen-

Information and Media Technologies 7(2): 601–613 (2012)
reprinted from: IPSJ Transactions on Advanced Computing Systems 5(2): 10–22 (2012)
© Information Processing Society of Japan

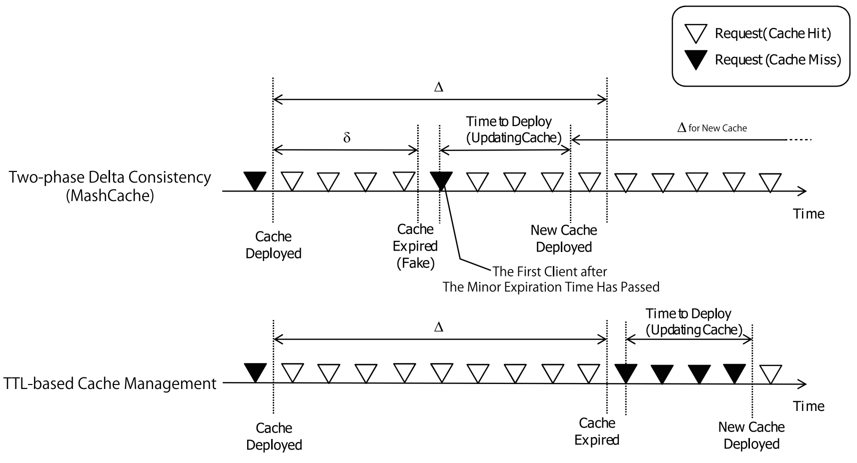
Request(Cache Hit)
Request (Cache Miss)
Δ
δ
Time to Deploy (Updating Cache)
Δ for New Cache
Two-phase Delta Consistency (MashCache)
Time
Cache Deployed
Cache Expired (Fake)
New Cache Deployed
The First Client after The Minor Expiration Time Has Passed
TTL-based Cache Management
Cache Expired

**Fig. 3** Cache updating in MashCache (above) and a general TTL-based system (below).

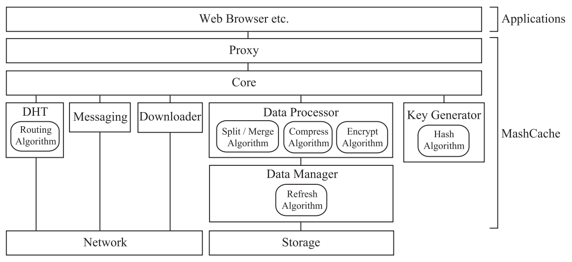

**Fig. 4** Architecture of MashCache.

erates CMD and chunks from the original contents, and restores them from the received CMD and chunks. Specifically, the data processor component offers functions such as splitting and merging, compression and decompression, and encryption and decryption. When these processes require additional information, we can use CMD to put it.

The data manager manages a lifetime of CMD and chunks. The data manager saves the CMD or the chunks, and set their expiration time when the client receives them. It manages the two types of expiration time for CMD or chunk; the minor expiration time and the major expiration time. The data manager keeps the data until it is expired.

The key generator creates keys for searching CMD and chunks. The key generator generates a unique string from queries for the content and uses it as a key of CMD. It also generates keys of each chunk. We use a hash algorithm to guarantee the keys do not collide without any central management servers and hide information of original queries.

## 4.1 Procedures

In advance, MashCache clients form an overlay network with the DHT to manage clients and caches of contents. New comers join to the overlay using a bootstrap client which have already joined to it. On the overlay, each client manages the caches of contents using keys generated from the queries which identify the contents. In MashCache, each cache of the contents is treated as CMD and some chunks; the chunks are fragments of content and the CMD contains keys of the chunks.

**Figure 5** shows a flow where a client requests a content. When the client issues queries for the content, the proxy receives them. The key generator generates a key to get CMD of the content

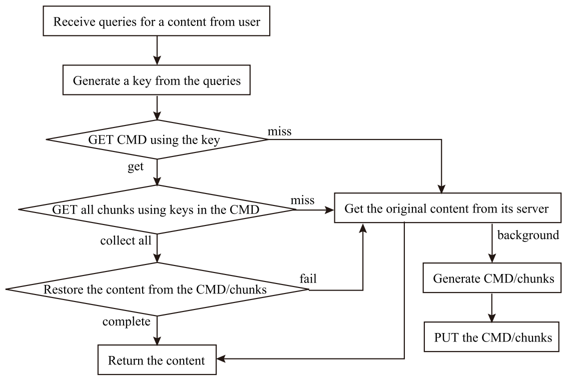
Receive queries for a content from user
Generate a key from the queries
GET CMD using the key
miss
get
GET all chunks using keys in the CMD
miss
Get the original content from its server
collect all
background
Restore the content from the CMD/chunks
fail
Generate CMD/chunks
complete
PUT the CMD/chunks
Return the content

**Fig. 5** Flow of requesting a content.

from the queries. The client uses the key to find a client which has the CMD from the DHT. And then the client also tries to get chunks of the content using keys which the CMD contains. When the client collects all of the chunks, it restores the content using the CMD and the chunks. However, if the client misses the CMD or some of the chunks or fails restoration, the client issues the queries to its original server and gets the content. The client which gets the content from the server generates CMD and chunks and puts them into the overlay. We describe how they get and put CMD and chunks in following sentences.

### 4.1.1 Getting CMD and Chunks

When a client *A* tries to get data (CMD or chunk), at first *A* finds a client *X* in charge of its key from the DHT, and then *A* sends a GET message which contains the key to *X*. **Figure 6** shows how *X* treats the GET message. *X* searches its data manager with the key for the data. *X* returns the data if it exists and does not expire, otherwise return a MISS message which informs *A* that the data is unavailable. Then, if its major expiration time has passed, the data manager removes the data. If its minor expiration time has passed, the data manager extends the minor expiration time by a minor delta $\delta$. This mechanism of *Two-phase Delta Consistency* enables MashCache to keep data missing low and refresh data frequently. On the other hand, a usual single expiration time causes burst data missing. We show this difference in Section 5.3.2.

### 4.1.2 Putting CMD and Chunks

When a client *B* puts data (CMD or chunk), at first *B* tries to

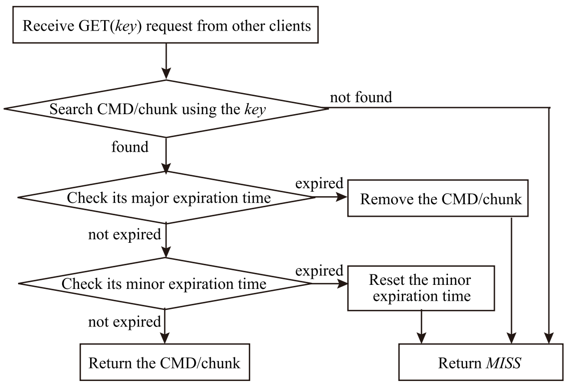
Receive GET(*key*) request from other clients
Search CMD/chunk using the *key*
not found
found
Check its major expiration time
expired
Remove the CMD/chunk
not expired
Check its minor expiration time
expired
Reset the minor expiration time
not expired
Return the CMD/chunk
Return *MISS*

**Fig. 6** Flow of getting CMD/chunk.

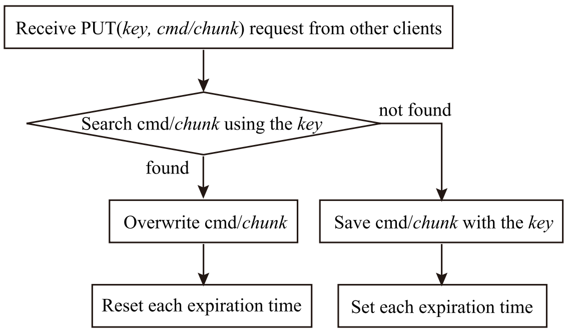
Receive PUT(*key, cmd/chunk*) request from other clients
Search cmd/*chunk* using the *key*
not found
found
Overwrite cmd/*chunk*
Save cmd/*chunk* with the *key*
Reset each expiration time
Set each expiration time

**Fig. 7** Flow of putting CMD/chunk.

find a client $Y$ in charge of its key from the DHT as the getting procedure, and then $B$ sends a PUT message which contains a set of the keys and the data to $Y$. **Figure 7** shows how $Y$ treats the PUT message. $Y$ searches its data manager with the key for the data. $Y$ saves the data with the key and sets its minor expiration time as $\delta$ later and major expiration time as $\Delta$ later. If $Y$ finds that the key already exists, $Y$ overwrites the data and resets the expiration times in the same way.

## 5. Evaluation

In this section, we describe the experiments which demonstrate MashCache mitigates flash crowds. To do the experiments, we implemented a prototype of MashCache using OverlayWeaver-0.9.9, a toolkit for developing overlay networks [6]. To confirm the efficiency of MashCache, we conducted the experiments with up to 2,500 emulated clients. We used two physical machines for the experiments. One is for a client emulator with a quad core 3.00 GHz CPU (Intel Xeon), a 12 GB RAM, a gigabit ethernet, and running Linux-2.6.20. The other is for a content server with a dual core 2.40 GHz CPU (Intel Core2Duo), a 2 GB RAM, a gigabit ethernet, and running Linux-2.6.31 and Apache HTTP server 2.2.12 [11].

We used the *Meridian* data set to emulate the latencies between clients. The data set is used by the Meridian project [12] and contains pair-wise RTT measurements between 2,500 DNS servers whose IP addresses are unique, spanning 6.25 million node pairs. It was collected on May 5–13, 2004. We set each clients' bandwidth to 10 Mbps.

We used OverlayWeaver's implementation of *Chord* [13] for the DHT and *MD5* hash algorithm [14] for the key generator. We configured maximum chunk size as 256 KB, and the cache expiration time $\delta$ and $\Delta$ to 5 seconds and 10 seconds, respectively.

To demonstrate that MashCache can actually deal with flash crowds on mashups, we simulated small flash crowds on a simple typical mashup service. In this experiment, we prepared a mashup service which consisted of three web servers named $M$, $A$ and $B$; $M$ provided 1 KB of data as a mashup service, and $A$ and $B$ provided 1 KB and 100 KB of data as external services. The mashup treated a client request in the following process; 1) $M$ received a client request for the mashup service, 2) $M$ fetched external contents from $A$, 3) $M$ generated contents using the external contents from $A$ and sent it back to the client, 4) the client got the rest of contents from $B$ to complete the mashup contents. When the maximum number of clients already established their connection with the servers, following clients got HTTP 503 error. The client which failed to get the contents continued to retry at 1,000 msec intervals until it completed its request.

In following section, we evaluate MashCache in three angle. First, we demonstrate that MashCache actually mitigates flash crowds on the web servers. Second, we evaluate that MashCache has enough scalability under flash crowds. Finally, we also show that MashCache can keep caches new with the load on the web servers stable in low level.

### 5.1 Mitigating Flash Crowds

MashCache can mitigate flash crowds even when the scale of flash crowds exceeds the capacity of web servers. In this section, we demonstrate that MashCache could absorb flash crowds and effectively deal with all requests from the clients as ideal mighty web servers.

Due to the limitation of our simulator, we scaled down the number of the clients to 100 nodes and the capacity of all the web servers to simulate flash crowds; we limited the bandwidth and the maximum number of established connections of these servers so that they failed down if all the clients rushed to them.

To confirm how effectively MashCache deals with flash crowds, we prepared two conditions, a simple server condition as reference *Direct* and an ideal server condition as reference *Oracle*. Direct means that MashCache is disabled. Oracle does not have any constraint on the maximum number of established connections of the web servers, i.e., all clients can directly get the contents from the servers even in flash crowds. Note that Oracle is not a practical condition since it assumes that we can successfully estimate the scale of flash crowds and the time flash crowds occur.

In this experiment, we measured two metrics: *request success rate* and *request receiving rate*.

- *Request success rate* represents the number of requests which successfully get the contents from web servers. If request success rate is higher even with high request rate, we can say that more clients can get the contents in flash crowds. We also measured "incomplete" request success rate, which means a client can only get a part of the contents in the mashup environment.
- *Request receiving rate* represents the number of requests which are directly treated by a web server per second. If request receiving rate is stable in low level in spite of flash

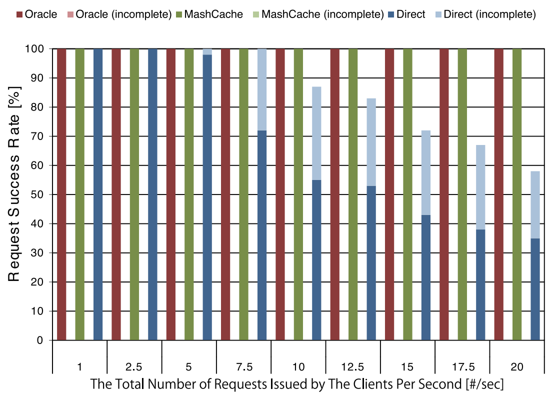

**Fig. 8** Request success rate with the total number of the requests issued by the clients per second.

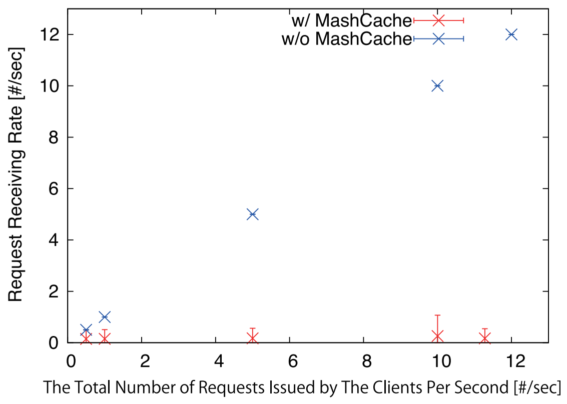

**Fig. 9** Request receiving rate with the total number of request issued by the clients per second.

crowds, the load of the web server is absorbed by MashCache well.

### 5.1.1 Request Success Rate

**Figure 8** shows MashCache keeps request success rate high under flash crowds. X-axis represents the total number of requests issued by the clients per second and Y-axis represents the request success rate. Note that "incomplete" means a client could get the contents from *M* but it could not get the contents from *B*.

The result demonstrated that MashCache could completely deal with all the client requests as Oracle did even when a client request rate increased. This is because MashCache treated most of requests without direct access to the web servers and the requests did not reach the web servers. In MashCache, Pure P2P Structure, Query Origin Key and Cache Meta Data enable clients to find caches they want without direct access to the web servers. On the other hand, many requests were dropped in Direct. In this experiment, the servers dropped 2% requests when the request rate was 5 req/sec and 65% requests when the request rate was 20 req/sec, respectively. This is because all requests issued by the clients rushed to the web servers and the number of the requests easily exceeded the capacity of the web servers.

### 5.1.2 Request Processing Rate on Web Servers

MashCache treats most of client requests instead of the web servers and MashCache keeps the load of the web servers low. To confirm this, we compared request receiving rate with MashCache and that without MashCache.

**Figure 9** shows how MashCache reduces the load on the web server. X-axis represents the total number of requests issued by clients per second and Y-axis represents request receiving rate. Although the load of the web server without MashCache increased linearly with the number of issued requests, that with MashCache was maintained virtually constant. When 10 requests were issued per second, request receiving rate with MashCache was about 98% smaller than that without MashCache. This result demonstrates that MashCache keeps client requests to be distributed to other clients on the MashCache overlay even if the issued requests are increasing. This is because MashCache restricts connecting to the web server directly to the first client after the minor expiration time expires and other clients use existent caches until new caches get available or the major expiration time expires.

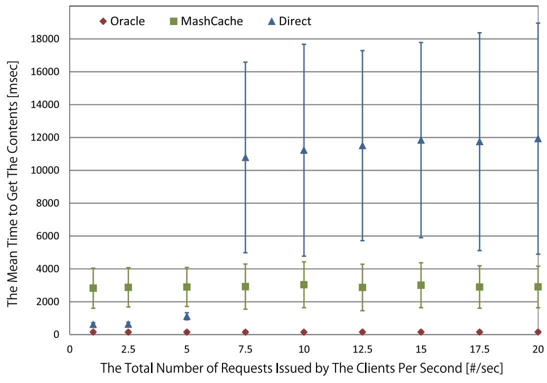

**Fig. 10** The mean time to get the contents with the total number of requests issued by the clients per second. An error bar represents a standard deviation.

## 5.2 Scalability of MashCache

MashCache can constantly perform even when incoming request rate increases or many clients participate the MashCache network. MashCache achieves that by Pure P2P Structure and Query Origin Key. In this section, we demonstrate that the time for getting contents did not increase when incoming request rate heated up, and did moderately increase when a huge number of clients use MashCache. We measured the time for getting contents in two angles: *incoming request rate* and *number of clients*. Incoming request rate represents the number of requests which try to get the contents. An ideal system can keep the time for getting contents low even if incoming request rate is high. Number of clients represents the number of clients which participate in MashCache's network. An ideal system do not rapidly increase the time for getting contents time with the number of clients.

### 5.2.1 Incoming Request Rate

MashCache manages caches and clients with decentralized ways, and that provides stable performance even when a huge number of requests are issued to the web servers. To show that MashCache can provide caches in such situation, we measured how long time each client took to get the contents. In this experiment, we simulated 100 clients.

**Figure 10** shows that MashCache keeps the time to get the contents constant even when a request rate increases. X-axis represents the total number of requests issued by the clients per second and Y-axis represents the time to get the contents. While MashCache and Oracle keep the time stable as about 2,915 msec and 152 msec respectively, the time in Direct increased from 1,116 msec to 10,786 msec when the request rate becomes over 7.5 req/sec. This is because many clients failed to get the contents and retried again in Direct. The retrying requests made the web server more congested. The deviation is ballooned because the time to get the contents differs widely whether the clients get the contents on the first time or not. On the other hand, MashCache, there is no client brings such congestion on the web servers.

The result also shows that MashCache has overheads to get the contents. In this experiment, the average time of MashCache is 2,764 msec larger than that of *Oracle*. Our analysis reports that most of the overhead is the lookup latency on DHT. However, it is possible to reduce the latency because the prototype of MashCache employs an original Chord, which has a large lookup latency reported in Ref. [15], as a DHT. We believe that the overhead can decrease to a realistic time using many well-known techniques, i.e., latency-aware DHT [15], [16], locality-aware replication [17], and etc.

### 5.2.2 Number of Clients

We designed MashCache to deal with flash crowds, so it should efficiently manage caches and clients even when a huge number of clients are coming. To confirm that MashCache has the scalability to a huge number of clients, we varied the number of clients on the MashCache overlay and measured getting contents time. In this experiment, we changed the number of clients from 10 to 2,500.

To clear up the theoretical limitation of the scalability, we first calculated the theoretical value of the mean of getting contents time $\bar{t}_{\text{cache}}$ by the below formulas [*1];

$$\bar{t}_{\text{search}} = \bar{l} \log n \quad (1)$$

$$\bar{t}_{\text{cmd}} = \bar{t}_{\text{search}} + \bar{l} + \frac{V_{\text{cmd}}}{v} \quad (2)$$

$$\bar{t}_{\text{chunk}} = \bar{t}_{\text{search}} + \bar{l} + \frac{\frac{V_{\text{chunk}}}{v}}{\left\lceil \frac{V_{\text{content}}}{V_{\text{chunk}}} \right\rceil} \quad (3)$$

$$\bar{t}_{\text{cache}} = \bar{t}_{\text{cmd}} + \bar{t}_{\text{chunk}} \quad (4)$$

whereas $n$ is the number of the clients, $\bar{l}$ is the mean latency of the Meridian data set, $v$ is the data transfer rate of a link, $V_*$ is the data size of $*$, $\bar{t}_{\text{search}}$ is the mean time to find a client bound to a key, $\bar{t}_{\text{cmd}}/\bar{t}_{\text{chunk}}/\bar{t}_{\text{cache}}$ are the time to get or deploy CMD/chunk/cache. The line named *Theoretical* in **Fig. 11** represents $\bar{t}_{\text{cache}}$. As the formula and Fig. 11 show, $\bar{t}_{\text{cache}}$ is $O(\log n)$. Note that $O(\log n)$ is the number of nodes that must be contacted to find a successor in an $n$-node Chord network, which is implemented in the prototype of MashCache as a DHT. This is because $n$ is involved in only $\bar{t}_{\text{search}}$ as we showed above, and MashCache's features prevent a particular client from becoming a hotspot.

Figure 11 shows the changes to get the contents with the number of MashCache clients. X-axis, in a logarithmic scale, represents the number of clients which participate in the overlay of MashCache, and Y-axis represents the mean time to get the contents with MashCache. In Fig. 11, the plots named *Experimental* represent the measurements of the time with the prototype

[*1] We consider just about the latencies and the data transfer time in this analysis.

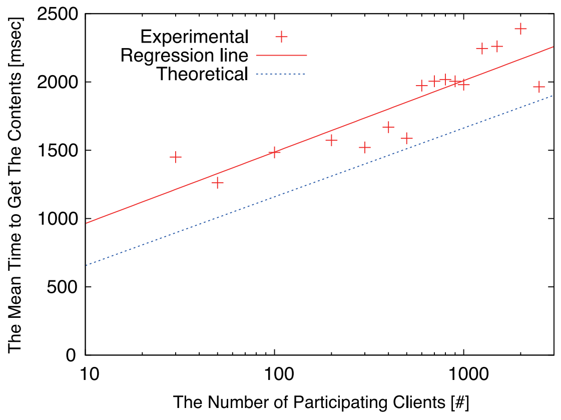

**Fig. 11** The mean time to get the contents with the number of MashCache nodes. Note that X-axis is in a logarithmic scale.

of MashCache in the simulation. The regression line approximates the plots by least-square regression using logarithmic values. The R2 coefficient of determination was 0.765. Thus, the result demonstrates that the time with the prototype of MashCache approximately increases in $O(\log(n))$ as calculated line. Note that the overhead of our simulator is included in the measurements. That is why the regression values were about 440 msec larger than the calculated values.

## 5.3 Cache Consistency

Cache based load balancing systems generally face to a problem how to maintain the consistency of caches especially for large scale systems. MashCache can keep them fresh with low load by Aggressive Caching and Two-phase Delta Consistency. In this section, we demonstrate that MashCache can frequently update caches without heavy loads on web servers through another simple experiment. In this experiment, we prepared one web server which had 1 MB of data as a static content and each clients issued requests for the content, and we emulated 100 clients which issued requests within specified periods. We measured the following two metrics; *the age of cache* and *load of cache updating*.

- *The age of cache* represents the elapsed time since a cache is created. If each cache is well maintained, the average age of caches takes low value. We started to count the age of a cache since its CMD was created.
- *Load of cache updating* represents how many requests reach to the original web servers when caches has expired. If cache update results in a low number of direct accesses to the web server, it is a good condition for the web server because the load of the web server is also low.

### 5.3.1 The Age of Cache

MashCache can keep caches new by frequent updates and most of clients can get new caches. Keeping caches for long time is one of effective ways to hold down the number of requests a web server treat, but this also involves a problem that the number of clients which get old caches increase and they cannot notice even if original contents are updated. In this section, we demonstrate that MashCache can keep the average age of caches low and reduce the number of requests the web server treat. In this experiment, we set $\delta$ and $\Delta$ to 5 seconds and 10 seconds, respectively when Two-phase Delta Consistency is enabled, and we set $\Delta$ to

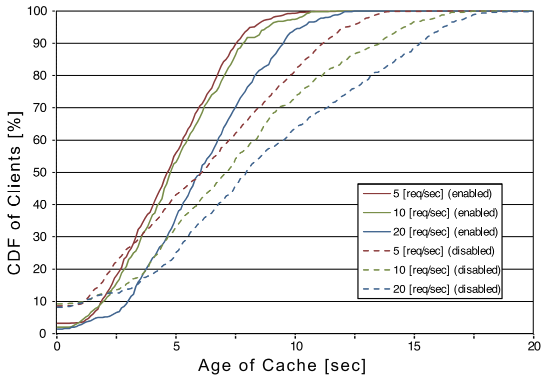

**Fig. 12** CDF of clients with cache age.

10 seconds when Two-phase Delta Consistency is disabled.

**Figure 12** compares the freshness of caches a client fetched with *Two-phase Delta Consistency* enabled and that disabled. X-axis represents the age of a cache a client received, and Y-axis represents the cumulative distribution function (CDF) of the number of clients. Each line represents the number of requests issued by the clients per second. A zero-aged-cache means that a client fetched the original contents from the web server and the cache is just created.

The result demonstrates that MashCache with Two-phase Delta Consistency enabled keeps many caches fresh even with low rate of requests received by the web servers. For example, when the request issuing rate was 10 req/sec, 90 percentile of all caches a client fetched with Two-phase Delta Consistency enabled and disabled were 7.9 sec and 13.6 sec, respectively. And the fractions of clients got a zero-aged-cache with Two-phase Delta Consistency enabled and disabled were 2.0% and 9.2%, respectively.

The reason why MashCache can keep caches fresh is that Two-phase Delta Consistency enables MashCache to effectively and periodically update caches. If Two-phase Delta Consistency is disabled, a number of clients try to establish a connection with a web server at the same time to fetch the cache-expired contents and the server will be overloaded in flash crowds. To avoid a web server down without Two-phase Delta Consistency, MashCache has to extend the time-to-live of a cache based on the scale of flash crowds. On the other hand, if Two-phase Delta Consistency is enabled, MashCache does not have to extend the time-to-live of a cache, which is the minor expiration in Two-phase Delta Consistency, because MashCache continues providing the caches whose major expiration time has not expired instead of a web server until the up-to-date caches are enough distributed to the MashCache overlay. The major expiration time ensures that a client gets a cache whose age is under $\bar{t}_{\text{cache}} + \Delta$.

We also found another effect of Two-phase Delta Consistency; the rate of clients get a zero-aged-cache decreases with increasing request rate. This fact means that the load of the web server does not rise even when a huge number of requests are issued. This is because Two-phase Delta Consistency causes only periodical cache updates despite changing of request rate. Therefore, MashCache can work enough under extreme flash crowds in which request rate changes variously.

### 5.3.2 Load of Cache Updating

MashCache can update caches frequently with the number of client accesses to a web server lowered. To confirm that, we measured the rate of requests a web server received when Two-phase Delta Consistency is enabled and disabled. In this experiment, we set $\delta$ and $\Delta$ to 5 seconds and 10 seconds, respectively when Two-phase Delta Consistency is enabled, and we set $\Delta$ to 5 seconds when Two-phase Delta Consistency is disabled.

**Figures 13** and **14** show the rate of requests the web server received and the number of requests issued by clients per second with Two-phase Delta Consistency disabled and enabled, respectively. X-axis represents elapsed time and Y-axis represents the number of requests per second.

The result demonstrates that Two-phase Delta Consistency reduces the peak rate of requests a web server received. The average rate of requests with and without Two-phase Delta Consistency were 1 req/sec and 10 req/sec, respectively. The reason for the difference in the results is that Two-phase Delta Consistency prevents clients from rushing a web server to update caches. As shown in Fig. 13, without Two-phase Delta Consistency, the load of a web server spiked to the rate of all the client accesses in flash crowds, i.e., all the clients rushed the web server in the period. On the other hand, as shown in Fig. 14, with Two-phase Delta Consistency, the load of a web server was well controlled in low level. This is because the minor expiration time triggered a cache update of only one client and all caches were still available to keep the load of a web server moderate. Therefore, MashCache keeps the load of the web server low and the caches fresh.

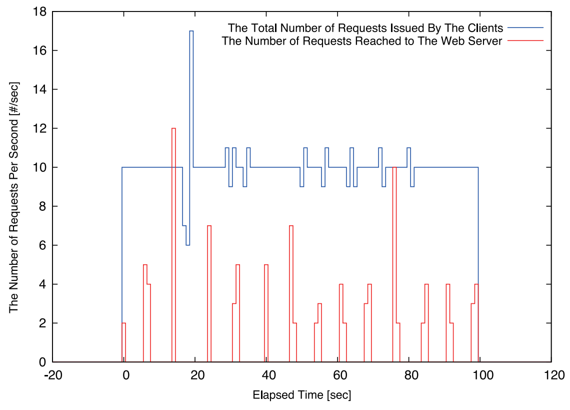

**Fig. 13** Request receiving rate without Two-phase Delta Consistency. Burst accesses occurred each time the caches expire.

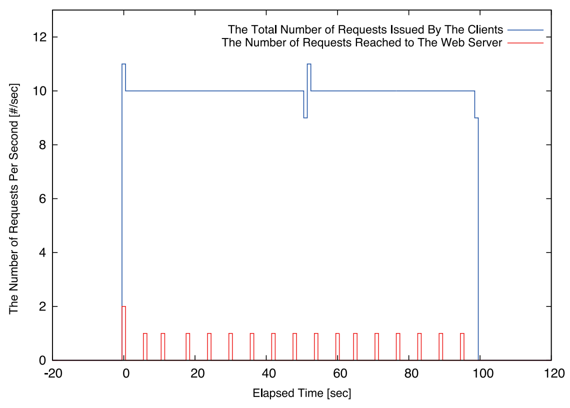

**Fig. 14** Request receiving rate with Two-phase Delta Consistency.

## 6. Discussion

### 6.1 Overhead

#### 6.1.1 Bandwidth Consumption

Consumption of network bandwidth increases for two reasons in MashCache. One is the communication to lookup or maintain DHT and the other is the data transfer caused to save caches on corresponding clients. The former is negligibly-small because the size of this kind of messages is tens of bytes and most of contents are larger than a few kilobytes. We can estimate the latter as follows. The expected value of bandwidth consumption for a client to receive responsible caches $E$ can be estimated by the following formulas;

$$\bar{c} = \frac{\bar{V}_{\text{content}}}{V_{\text{chunk}}} \cdot \frac{u}{n} \tag{5}$$

$$\bar{t}_c = \frac{V_{\text{chunk}}}{\bar{v}} \tag{6}$$

$$E = \frac{u}{n} \sum_{i}^{\bar{c}} {}_{\bar{c}}C_i \, i \left(\frac{\bar{t}_c}{T}\right)^i \left(1 - \frac{\bar{t}_c}{T}\right)^{\bar{c}-i}, \tag{7}$$

where $\bar{c}$ is the mean number of caches which each client holds, $\bar{V}_{\text{content}}$ is the mean size of web contents, $V_{\text{chunk}}$ is the size of chunk, $n$ is the number of the clients, $u$ is the number of unique queries which are issued by the clients simultaneously, $\bar{v}$ is the mean upload bandwidth of the clients, $\bar{t}_c$ is the mean time to transfer a chunk, and $T$ is TTL of each cache. Then, the expected value of total network bandwidth consumption to save the whole cache in MashCache is $nE$.

The formulas show that the bandwidth consumption to save caches increases in proportion to the square of the number of unique requests issued at a certain moment. For example, we consider a concrete situation expressed by the following variables $n$ = 1,000,000, $\bar{v}$ = 1 Mbps, $T$ = 5 sec, $V_{\text{chunk}}$ = 256 KB, $\bar{V}_{\text{content}}$ = 1 MB. In this situation, the worst case is that all of clients issue completely different requests simultaneously ($u = n$). In this case, the bandwidth consumption of each client $E$ is 0.8 Mbps and the total usage in the whole system $nE$ is 800 Gbps. This total usage is about 1% of average traffic per second of the Internet in 2010 [18].

However, the worst case, in which all the clients issue completely different requests simultaneously, is quite rare because popularity of web contents shows Zipf-like distribution [19]. With the increase of the number of coincident requests, $u$ gets small for $n$ and the bandwidth consumption by MashCache also gets small.

#### 6.1.2 Latency

DHT requires some hops to find a corresponding node for a target cache. So, MashCache clients get larger latency than usual direct access to web servers. Not in flash crowds, each client gets one-lookup-latency $\bar{t}_{\text{search}}$ because the client fails to look up CMD and tries to get an original content from a web server. Using the mean of inter-node latency $l$ = 75.809 msec [12], we can get $\bar{t}_{\text{search}}$ to be 1,000 msec when the number of node $n$ = 9,352, 1,500 msec when $n$ = 904,380, and 2,000 msec when $n$ = 87,458,292. Therefore, the increase of the latency caused by MashCache can be estimated from 1 sec to 2 sec. In addition, the latency decreases to about 250 msec if we use latency-aware DHT [16] to manage caches.

Then, we can estimate the increase of latency not in flash crowds is less than 1 sec when the depth of references of contents is three or less. The reference here means that a content requires other contents. For example, the depth of reference is three when foo.html contains a reference to bar.css and fiz.js, and bar.css contains a reference to baz.gif. MashCache tries to get caches in parallel to suppress the rise in latency. In the example, the client starts to get bar.css and fiz.js simultaneously. Therefore, the overhead of latency for $d$-depth contents can be estimated as $\bar{l}d$; therefore, the overhead of latency for the example is about 750 msec with the latency-aware DHT of which the mean latency is 250 msec.

MashCache works better with larger size objects because the latency gets smaller compared to the total time for getting objects. In MashCache, the fraction of the latency is about 10% or less when the size of object is a few MB. **Table 1** shows the estimated size of a content with the percentage of the latency in total time to get the content through 5 Mbps channel. For example, the overhead of the latency to get 1.41 MB content is about 10% when the mean time of lookup latency is 250 msec as in the latency-aware DHT [16]. We calculate the mean bandwidth as 5 Mbps because 78% of users of the Internet use 5 Mbps or less bandwidth network channel [20].

#### 6.1.3 Reducing the Overheads

Although MashCache involves the overhead of the bandwidth consumption caused by Aggressive Caching, the overhead of the latency can be reduced by appropriately switching from using caches to not-using caches to get objects. In Aggressive Caching, each client generates caches which have been expired or not existent, but this does not imply the enforcement of the use of the deployed caches. Not in flash crowds, each client can get contents directly from original web servers, and concurrently check, generate and deploy caches.

### 6.2 Security

#### 6.2.1 Privacy

We assume that each confidential cache should be encrypted by the client which generates and deploys the cache. MashCache

**Table 1** The estimated size of a content with the percentage of the latency in total time to get the content through 5 Mbps channel.

| Percentage of latency [%] | latency [msec] | | | | |
|---|---|---|---|---|---|
| | 76 | 250 | 1,000 | 1,500 | 2,000 |
| 0.1% | 47 MB | 156 MB | 624 MB | 937 MB | 1.22 GB |
| 1% | 4.70 MB | 15.5 MB | 61.9 MB | 92.8 MB | 124 MB |
| 10% | 438 KB | 1.41 MB | 5.6 MB | 8.44 MB | 11.3 MB |
| 25% | 146 KB | 469 KB | 1.88 MB | 2.81 MB | 3.75 MB |
| 50% | 48.6 KB | 156 KB | 640 KB | 960 KB | 1.25 MB |

Information and Media Technologies 7(2): 601–613 (2012)
reprinted from: IPSJ Transactions on Advanced Computing Systems 5(2): 10–22 (2012)
© Information Processing Society of Japan

enables us to protect privacy by the following three extensions.

- Encryption of caches by original request queries
- Encryption of caches prohibited from being shared by the content provider using a private-key of the client which get the contents
- Avoidance to upload caches to the MashCache's network when the content provider prohibits the clients from storing any cache

The first extension is that each client encrypts caches using original request queries as encryption keys. If the caches contain privacy of the client, the request queries contain unique information like cookies in many cases. Therefore, most of caches are protected from other clients which should not get the caches.

The second extension is that each client encrypts caches using a private-key of the client when the clients are prohibited from sharing by the content provider. For example, each client encrypts the caches when the Cache-Control header which is contained in the response headers of HTTP/1.1 or the HTML headers indicates prohibition of sharing the contents as caches (e.g., no-cache). This expansion is effective for the case like that the contents are access controlled by IP address. In this case, the first expansion is not effective because invalid clients can get the caches, but the second expansion prohibits these invalid client cannot get the caches. However, this way cannot protect the privacy if the prohibition is not correctly set. And this way might degrade the effectiveness of MashCache's load balancing if the prohibition is set for the contents which do not contain any privacy.

The third extension is that each client does not upload caches when the clients are prohibited from storing caches. For example, as with the second extension, each client does not upload the caches the headers indicate prohibition of storing the contents as caches (e.g., no-store). This expansion is stronger way to protect the privacy than the second extension. However, along with the second extension, the inadequate settings of prohibitions lead to the flow of private information or the insufficient load balancing. Note that these problems of the second/third extensions are common to load balancing techniques by caches like proxy servers or other CDNs.

The first extension and the second/third extensions are complementary each other. The first extension is effective to the contents for which requests differ depending on access authorities. In this case, the private contents is protected by encryptions even if the caches are shared on the overlay network of MashCache. On the other hand, the second extension protects the privacy regardless of request queries when the settings of cache sharing is correct. The third extension drops the caches and never upload them to the network for cache sharing regardless of request queries when the settings of the cache storing is correct.

#### 6.2.2 Integrity

It is difficult for MashCache to guarantee the integrity of the caches because the system depends on the client-side P2P network. However, in MashCache, it is easy for the contents providers to check defacing of their caches. The contents providers can easily spot the caches of their contents because MashCache uses the structured overlay network and we can find the place of each cache through DHT. By doing so, the period for which the caches are tampered with can be short.

**Table 2** The objects constitute the top page of amazon.co.jp.

| | w/ Cookie | | Whole | |
|---|---|---|---|---|
| | Size | Number | Size | Number |
| HTML | 40.6 KB | 4 | 40.6 KB | 4 |
| CSS file | 0 KB | 0 | 33.0 KB | 7 |
| JS file | 883 B | 2 | 82.5 KB | 10 |
| Image | 128 B | 2 | 356.5 KB | 72 |
| SWF file | 0 KB | 0 | 209.4 KB | 2 |
| Total | 41.6 KB (5.8%) | 8 (12.5%) | 722.1 KB | 95 |

### 6.3 Prospects of Cache Hit

To increase the cache hit rate, MashCache creates caches at the object level, not at the web page level. This means it caches objects each of which constitutes a web page. MashCache generates different caches when the queries for getting each object are different. Our approach is based on the fact that most of contents do not require cookies even if they are parts of personalized web pages. For example, **Table 2** shows how many objects in the top page of amazon.co.jp need cookies to get them. Even though this page is a personal page, about 90% of the objects do not require cookies. Therefore, we believe that MashCache allows us to share many caches generated from the request queries.

### 6.4 Consistency

MashCache may return inconsistent caches with the original contents in the following two situations. One is that the contents are renewed after MashCache caches them. To address this problem, MashCache employs a TTL cache management and Two-phase Delta Consistency. Both techniques allows us to periodically update the cache, avoiding generating enormous requests to the servers, described in Section 3. In addition, MashCache guarantees that clients do not see caches in temporally-wrong order. This is because MashCache puts caches into the specified places on the structured overlay network defined by DHT. This feature also makes it easier to update the caches.

The other is that contents are generated based on the server's status. This case can be controlled as long as contents providers adequately set the Cache-Control headers defined in HTTP. For example, the contents providers should set the Cache-Control header to prohibit the clients from using the caches when they use time or random values to generate contents. If the headers are not set, MashCache caches such contents and provides users caches that is not the original ones. However, this problem occurs when we employ cache mechanisms such as proxy servers and Content Delivery Network.

## 7. Related Work

### 7.1 Load-Balancing with Cache Servers

Server-side approaches have been widely studied to handle flash crowds. The most basic way to reduce the load of a web server is to use a load-balancer and replica servers. The load-balancer spreads requests from clients across the replica servers and reduces the load of each replica server. Squid [21] is a proxy server which serves contents in place of their origin web server. Backslash [22] is intended as a drop-in replacement for web servers and reverse proxy caches. Backslash is built on a

DHT and each cache server is connected with overlay network. Content delivery networks (CDNs) also take this approach. In Akamai [23], which is one of the largest commercial services, many proxy servers are distributed worldwide and connected by special high-speed lines. In CoralCDN [24] and DOH [25], overlay networks are used instead of special high-speed networks.

The server-side approaches against flash crowds have two good points; the easiness of cache management and the transparency for clients. The former means that the contents providers can control which contents set as caches into the proxy servers, and the number of proxy servers for load balancing. The latter means that we can benefit from the approaches without any modification of clients.

The server-side approaches also have a weak point; the difficulty of estimating the appropriate number of cache servers. If the estimation is small, the servers cannot handle flash crowds, leading to the service stop. In fact, Akamai became unavailable because of an avalanche of requests [26]. On the other hand, when the estimation is big, the servers waste energy and costs while flash crowds do not occur. It is quite difficult to prepare the appropriate number of cache servers. Although client-side approaches need to modify the client, they release us to accurately estimate the servers. In the next section, we describe client-side approaches in details.

### 7.2 Cache Sharing between Clients

This approach uses clients' resources to deploy contents. This approach is excellent with heavy loads because it can use many resources when many clients try to get a content. However, this approach needs a dedicated node which modifies each contents to share and manages the modified contents and clients. Then this approach can be classified into two categories; one is server-based management approaches and the other is client-based management ones. In the former, a content provider manages the entire system. In the latter, the clients which get a content from its server collectively manage the entire system.

BuddyWeb [27], BitTorrent [3], Avalanche [4] and Flashback [5] are the server-based management approaches. In BuddyWeb, participating clients offer their local caches to others and the caches which each client has are managed by a special server. In the other three ways, a content server provides peaces of contents and each client treads and collects the peaces. That reduces a load of the origin server, but these ways are not effective in mashups. This is because a mashup is not provided from a single provider and a mashup provider cannot force its depending providers to use these ways.

Squirrel [28] and the technique of Linga et al. [29] are the client-based management approaches. These ways do not require content providers to modify servers or to manage clients and caches. Our solution is also this way. In Squirrel, each client allows other clients in same LAN to use its local caches. This is concerned with neither reducing the load of a web server nor cooperating with a large number of clients in the Internet, but accelerating getting contents [29] manages caches of web objects which each client gets from web servers. It keeps high cache hit rates and the network locality under *churn* in which many clients arrive and departure extremely frequently. However, it is not concerned with flash crowds in which we have to consider how to manage loads, and we also have to consider cache freshness even if in the situations many clients rush on the contents.

## 8. Conclusion

In this paper, we proposed *MashCache*, a scalable client-side architecture for taming harsh effects of flash crowds. The design of MashCache is derived from good features of flash crowds. Flash crowds reach the peak within more than tens of seconds, flash crowds last for a short period, and large fraction of requests concentrate into small fraction of contents. MashCache handles flash crowds by the following four features: the Pure P2P Structure, the Aggressive Caching, the Query Origin Key and the Cache Meta Data. The pure P2P structure, the Aggressive Caching and the Query Origin Key release us from hard work to predict when and how large flash crowds occurs. The Cache Meta Data enables us to avoid producing hot spots in MashCache nodes. In addition, the Two-phase Delta Consistency enables MashCache to update caches frequently without high load on web servers. Our experimental results show that MashCache reduces the number of requests received by a web server using shared caches and the required time to get the caches increases in $O(\log n)$. Also, MashCache updates the caches every few seconds without spikes of access to the web server.

### Reference

[1] Jung, J., Krishnamurthy, B. and Rabinovich, M.: Flash Crowds and Denial of Service Attacks: Characterization and Implications for CDNs and Web Sites, *Proc. Int'l Conference on WWW*, pp.293–304 (2002).

[2] Freedman, M.J.: Experiences with CoralCDN: A Five-Year Operational View, *Proc. USENIX Symposium on Networked Systems Design and Implementation*, pp.7–7 (digital) (2010).

[3] Cohen, B.: Incentives Build Robustness in BitTorrent, *Proc. Int'l Workshop on Peer-to-Peer Systems* (2003).

[4] Gkantsidis, C., Miller, J. and Rodriguez, P.: Comprehensive view of a live network coding P2P system, *Proc. ACM Special Interest Group on Data Communications Conference*, pp.177–188 (2006).

[5] Deshpande, M., Amit, A., Chang, M., Venkatasubramanian, N. and Mehrotra, S.: Flashback: A Peer-to-Peer Web Server for Flash Crowds, *Proc. IEEE Int'l Conference on Distributed Computing Systems*, pp.15–15 (digital) (2007).

[6] Shudo, K., Tanaka, Y. and Sekiguchi, S.: Overlay Weaver: An overlay construction toolkit, *Computer Communications*, Vol.31, No.2, pp.402–412 (2008).

[7] SourceForge, Inc.: Slashdot, available from ⟨http://slashdot.org/⟩.

[8] Adler, S.: The Slashdot Effect: An Analysis of Three Internet Publications, available from ⟨http://hup.hu/old/stuff/slashdotted/SlashDotEffect.html⟩.

[9] Amazon Web Services LLC: Amazon Elastic Compute Cloud, available from ⟨http://aws.amazon.com/ec2/⟩.

[10] Google, Inc.: Google App Engine, available from ⟨https://appengine.google.com/⟩.

[11] The Apache Software Foundation: Apache HTTP Server, available from ⟨http://httpd.apache.org/⟩.

[12] Wong, B., Slivkins, A. and Sirer, E.G.: Meridian: A Lightweight Network Location Service without Virtual Coordinates, *Proc. ACM Special Interest Group on Data Communication Conference*, pp.85–96 (2005).

[13] Stoica, I., Morris, R., Karger, D., Kaashoek, M.F. and Balakrishnan, H.: Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications, *Proc. ACM Special Interest Group on Data Communications Conference*, pp.149–160 (2001).

[14] Rivest, R.: The MD5 Message-Digest Algorithm, *RFC1321* (1992).

[15] Dabek, F., Li, J., Sit, E., Robertson, J., Kaashoes, M.F. and Morris, R.: Designing a DHT for low latency and high throughput., *Proc. USENIX Symposium on Networked Systems Design and Implementa-*

Information and Media Technologies 7(2): 601–613 (2012)
reprinted from: IPSJ Transactions on Advanced Computing Systems 5(2): 10–22 (2012)
© Information Processing Society of Japan

*tion*, pp.85–98 (2004).

[16] Kaune, S., Lauinger, T., Kovacevic, A. and Pussep, K.: Embracing the Peer Next Door: Proximity in Kademlia., *Proc. IEEE Int'l Conference on Peer-to-Peer Computing*, pp.343–350 (2008).

[17] Locher, T., Schmid, S. and Wattenhofer, R.: eQuus: A Provably Robust and Locality-Aware Peer-to-Peer System, *Proc. IEEE Int'l Conference on Peer-to-Peer Computing*, pp.3–11 (2006).

[18] Cisco: Cisco Visual Networking Index: Forecast and Methodology, 2010–2015, Technical report, Cisco (2010).

[19] Breslau, L., Cao, P., Fan, L., Phillips, G. and Shenker, S.: Web caching and Zipf-like distributions: Evidence and implications, *Proc. IEEE International Conference on Computer Communications*, pp.126–134 (1999).

[20] Pingdom: The REAL connection speeds for Internet users across the world (charts), Technical report (2010), available from ⟨http://royal.pingdom.com/2010/11/12/real-connection-speeds-for-internet-users-across-the-world/⟩.

[21] Wessels, D., Nordstrom, H., Rousskov, A., Chadd, A., Collins, R., Serassio, G., Wilton, S. and Francesco, C.: Squid: Optimising Web Delivery, available from ⟨http://www.squid-cache.org/⟩.

[22] Stading, T., Maniatis, P. and Baker, M.: Peer-to-Peer Caching Schemes to Address Flash Crowds, *Proc. 1st Int'l Workshop on Peer-to-Peer Systems*, pp.203–213 (2001).

[23] Akamai Technologies: Akamai, available from ⟨http://www.akamai.com/⟩.

[24] Freedman, M.J., Freudenthal, E. and Maziéres, D.: Democratizing content publication with Coral, *Proc. USENIX Symposium on Networked Systems Design and Implementation*, pp.18–18 (digital) (2004).

[25] Jernberg, J., Vlassov, V., Ghodsi, A. and Haridi, S.: DOH: A Content Delivery Peer-to-Peer Network, *Proc. European Conference on Parallel Computing* (2006).

[26] Akamai Technologies: Akamai Provides Insight into Internet Denial of Service Attack, available from ⟨http://www.akamai.com/html/about/press/releases/2004/press_061604.html⟩.

[27] Wang, X., Ng, W., Ooi, B., Tan, K.-L. and Zhou, A.: BuddyWeb: A P2P-Based Collaborative Web Caching System, *Proc. Int'l Workshop on Peer-to-Peer Computing*, pp.247–251 (2002).

[28] Iyer, S., Rowstron, A. and Druschel, P.: Squirrel: A decentralized peer-to-peer web cache, *Proc. ACM Symposium on Principles of Distributed Computing*, pp.213–222 (2002).

[29] Linga, P., Gupta, I. and Birman, K.: A Churn-Resistant Peer-to-Peer Web Caching System, *Proc. ACM Workshop on Survivable and Self-Regenerative Systems*, pp.1–10 (2003).

**Hikaru Horie** was born in 1985. He received his B.E. and M.E. degrees from Keio University in 2009 and 2011, respectively. He is currently a Ph.D. candidate in School of Science for Open and Environmental Systems at Keio University and the president of Lightmarks Corp. he established, since 2011. His research interests include overlay networks and cloud computing. He is a member of IPSJ and ACM.

**Masato Asahara** received his B.E. degree from the University of Electro-communicatsions in 2005, M.E. degree and Ph.D. degree from Keio University in 2007 and 2011, respectively. Since 2010 he has worked at NEC Service Platforms Research Labs. His research interests include Peer-to-Peer systems, distributed systems, middleware and operating systems. He is a member of IPSJ, IEEE/CS, ACM and USENIX.

**Hiroshi Yamada** was born in 1981. He received his B.E. and M.E. degrees from the University of Electro-communications in 2004 and 2006, respectively. He received his Ph.D. degree from Keio University in 2009. He is currently a research associate of the Faculty of Science and Technology at Keio University. His research interests include operating systems, virtualization, and dependable systems. He is a member of ACM, USENIX and IEEE/CS.

**Kenji Kono** received his B.Sc. degree in 1993, M.Sc. degree in 1995, and Ph.D. degree in 2000, all in computer science from the University of Tokyo. He is an associate professor of the Department of Information and Computer Science at Keio University. His research interests include operating systems, system software, and Internet security. He is a member of the IEEE/CS, ACM and USENIX.