Vol.2011-OS-116 No.1
2011/1/24

MashCache: Flash Crowds 耐性を持つマッシュアップサービス実現手法

堀江 光[†1] 浅原理人[†1]
山田浩史[†1] 河野健二[†1]

ウェブサービスにおける応答遅延や異常停止の一因として Flash Crowds と呼ばれる突発的なアクセス集中が知られている． Flash Crowds は発生時期や規模の予測が困難なため，正確な予測を前提としない対策を要する．また，近年のウェブサービスには提供者の異なるサービスを複数組み合わせたマッシュアップが増加しているが，各サービスにおいて十分な Flash Crowds 対策を講じることは運用ポリシー等の違いから困難である．本論文では，クライアント資源を利用することで Flash Crowds への耐性を持ち，コンテンツの提供元が多岐に渡る場合にも対応可能なコンテンツ配信手法として *MashCache* を提案する． MashCache はクライアント間で形成する Peer-to-Peer (P2P) ネットワーク上でキャッシュを管理・共有することでサーバへのアクセスを減少させる．この設計は Flash Crowds の詳細な分析に基づいている．実インターネットで計測されたネットワーク遅延と最大で 2,500 台の仮想クライアントを用いた実験では，サーバへのリクエスト数を約 98 % 減少させることを確認した．

MashCache: Making Mashup-Services Resistant to Flash Crowds

Hikaru Horie,[†1] Masato Asahara,[†1]
Hiroshi Yamada[†1] and Kenji Kono [†1]

Mashup services are widely used to provide rich and complex services. Making mashup services resistant to flash crowds is quite difficult for the following reasons. First, it is almost impossible to prepare replica/cache servers in advance to handle flash crowds because the scale and the time of flash crowds are unpredictable. Second, the providers of mashup services can not enforce the use of flash-crowd-resistant mechanisms on all the external services on which the mashup services depend. Considering these circumstances, the use of clients' resources is attractive for mashup services. The total resources naturally increase when flash crowd occurs, and no server-side mechanism is needed to handle flash crowds. In this paper, we describe design and implementation of *MashCache*, a practical approach to making mashups resistant to flash crowds. MashCache is a peer-to-peer (P2P) based network that works on the client-side, and carefully tailored to handle flash crowds for mashup services. In our experiments with up to 2,500 emulated clients, a prototype of MashCache reduces the number of requests to the original web servers by about 98 % with moderate overheads.

1. はじめに

近年ウェブサービスは様々な場面で利用されるようになり重要性が高まっているが，一方で，サービス障害による被害も増大する傾向にある．例えば Amazon.com[1)] のサービスが 1 時間利用不能になると $200,000 の損害が発生するとの試算もあり，堅牢なサービスを構築することは極めて重要な課題である[2)]．しかし，多くのウェブサービスは Flash Crowds と呼ばれる突発的なアクセス集中によってしばしば不安定となり，サービスの応答遅延や異常停止といった障害に陥る．サービス提供者は Flash Crowds 発生時の障害発生を防ぐためにサーバ資源を増強する等の対策を取っているが，Flash Crowds はその原因が多岐に渡るために発生時期や規模の予測が難しく，適切な対策を施すことは困難である．

また，マッシュアップが Flash Crowds 対策をより困難なものにしている．マッシュアップとは提供元の異なる複数のサービスを組み合わせることで実現された新しいサービスのことで，低い開発コストで高度なサービスを実現可能な事から近年広く用いられている．マッシュアップに対して Flash Crowds が起こった場合，依存している外部のサービスにもその負荷が伝播するため，外部のサービスで十分な対策が取られていなければ，そちらで障害が発生し得る．従って，マッシュアップにおいて Flash Crowds へ対処するには関係する全てのサービスで十分な対策を施す必要があるが，提供者の異なる各サービスは運用ポリシーも異なるため対策を強制することは困難である．

Flash Crowds の予測困難性とマッシュアップの制御困難性を考慮すると，次の理由からクライアント資源を利用する事が適している．まず，Flash Crowds が起こった時，対象となっているコンテンツを求めるクライアント数の増加につれてクライアント資源も増加する．クライアント資源が利用可能であれば，Flash Crowds 発生時期や規模を予測することなく Flash Crowds による負荷を軽減することができる．また，こういったクライアント資源の管理をクライアントの連携によって実現できれば，サーバ側に Flash Crowds 対策と

†1 慶應義塾大学
Keio University

しての機構を組み込む必要がなくなる．すなわち，依存する外部サービスが Flash Crowds 対策を一切施していない場合でもマッシュアップを安定的に配信することができる．

本論文では，マッシュアップでも利用可能な Flash Crowds 対策として MashCache を提案する． MashCache はクライアント資源をクライアント間で管理しキャッシュを共有するため，ウェブサーバを始めとしてサービス提供者側での対応は一切必要としない． Flash Crowds 発生時に負荷が一極集中することを避けるため Pure P2P 型の構造をとる．すなわち MashCache を利用する全てのクライアントは同等の機能を持つに扱う．また， MashCache の設計は Flash Crowds の詳細な分析に基づく．この分析には Jung ら[3] や Freedman[4] による実際に起こった Flash Crowds の報告を用いた．

提案手法の有用性を検証するために， MashCache のプロトタイプをオーバレイネットワーク構築ツールキット Overlay Weaver[5] を用いて実装した． MashCache がサーバの負荷を軽減することを確認するために最大 2,500 台の仮想クライアントを用いシミュレーションを行ったところ，サーバで処理するリクエスト数を約 98 % 減少させることを確認した．

本論文の構成は以下の通りである． 2 章では Flash Crowds の詳細な分析とそれに基づく Flash Crowds 対策の要件について述べる． 3 章では関連研究について述べる． 4 章では本論文の提案手法について説明する． 5 章では MashCache の実装について説明する． 6 章では評価実験について述べる．最後に 7 章で本論文をまとめる．

2. Flash Crowds の分析と対策

2.1 Flash Crowds の分析

Flash Crowds は発生時期や規模の予測が極めて困難である．これは Flash Crowds の発生が様々な事象に起因するためで，例えば事故や自然災害の他にも一般的なニュースや Slashdot 効果[6] 等によっても発生する．こうした Flash Crowds の予測の難しさは，過不足のない適切な対策を行うことの妨げになっている．一方で Flash Crowds はその取り扱いを容易にしうる性質も持ち合わせていることがわかった．本章では， Flash Crowds を詳細に分析し対策に必要な要素を明確化する．

2.1.1 ピーク到達までの時間

Flash Crowds 発生時，リクエスト数に増加の兆候が見られてからピークに到達するまでは少なくとも数十秒以上の時間を要する． Jung らは実運用されている 2 つのウェブサイトにて発生した Flash Crowds をアクセスログを用いて分析した[3]．一方の Flash Crowds では，毎秒数件だったリクエスト数が約 4,100 秒かけて毎秒 6,719 件のピークに到達した．もう一方では，毎秒数件だったリクエスト数が 40 秒かけて毎秒 610 件のピークに到達した．また， Freedman は CoralCDN[7] を 5 年間実運用して得たトラフィックログから，数秒間でアクセスが急増することは極めて稀であると報告している[4]． Flash Crowds を詳細に解析するため Freedman は対象とする期間をエポックという短い時間単位に分割し，各エポックにおけるアクセスの増加率を集計した．この分析で数秒間のうちにアクセスが何十倍にもなるような状況は稀であると報告している．

2.1.2 継続時間の短さ

Flash Crowds の継続時間は数分から数時間程度である． Jung らの分析によると，2 つのウェブサイトで観測した Flash Crowds の継続時間はそれぞれ 100 分と 282 分であった[3]．また， Freedman は CoralCDN を利用したウェブサイトのひとつで見られた Slashdot 効果の実態を報告している[4]．このとき対象のウェブサイトへの毎秒のリクエスト数は平常時より約 15,000 件増加しており，これはおよそ 3 時間継続した．

2.1.3 リクエストの局所性

リクエストの大部分は一部の限られたコンテンツに集中する． Jung らは Flash Crowds が発生したウェブサイトにおいて上位 10 % の人気コンテンツがリクエストの 90 % 以上を占めることを示した[3]. また， Freedman は CoralCDN を利用したリクエストの内訳を調べ，上位 0.01 % の 人気 URL に対するリクエストが全体のの 49.1 % を占め， さらに，上位 10 % までの URL に対するリクエストは全体の 92.2 % を占めることを示した[4]．

2.2 Flash Crowds 対策が考慮すべき事項

2.2.1 Flash Crowds の予測困難性

2.1 で述べたように Flash Crowds は発生の時期や規模の予測が困難であるため，正確な予測を前提としない対策が必要である． Flash Crowds に対処するには負荷がピークに達する前にそれに耐えうる資源を準備する必要があるが，いつ起こるかわからない Flash Crowds のために必要量の数千倍の資源を稼働させておくことは運用コストの面から不適である．また，見積もりが甘かった場合には障害の発生につながってしまうという問題もある．

一方，近年では Amazon EC2[8] に代表されるクラウド環境の登場により，使用する資源の量を負荷に応じて柔軟に拡縮することが可能になっているが， Flash Crowds の対策として完全ではない．なぜなら，クラウド環境を提供するホストマシン群の数が限られていることやあらかじめ指定した予算内でしか拡縮しないことなどから， Flash Crowds のように突発的な意図しない負荷増への対応が難しいためである．また，このあと 2.2.2 で述べるように，クラウド環境の利用といったサービス提供者側での対応を要する対策ではマッシュ

アップに対応できないという問題もある．

2.2.2 マッシュアップの制御困難性

マッシュアップでは，関係する全ての外部サービスにおいて適切に対応することは困難であるため，各サービス提供者による対応を必要としない対策が必要である．なぜなら，マッシュアップ提供者にとって依存するサービスにおける障害の発生が痛手である一方で，これらのサービス提供者に対して Flash Crowds 対策を講じることを強制できないためである．また，マッシュアップが依存する外部のサービスはそれぞれ運用ポリシーも異なることが多く，仮に全てのサービス提供者が Flash Crowds 対策を講じたとしても対応可能な規模がそれぞれ異なるといった問題が生じる．

2.2.3 負荷の不均一性

クライアント資源を用いキャッシュを配信する手法を採る場合，各クライアントの負荷をできるだけ均一化する必要がある．ウェブサービスで用いられるコンテンツはサイズも多様化しており，例えば HTML ファイル，スタイルシート，JavaScript 等のテキストデータは数 KB から数十 KB 程度であるが，画像や音楽ファイルは数百 KB から数 MB，動画に至ってはそれ以上になることも多い．これらをそのまま単一のキャッシュとして扱うと，サイズの大きいキャッシュや人気の高いキャッシュの配信に関わるクライアントの負担が極端に高くなるだけでなく，そこがボトルネックとなることで全体の性能に悪影響影響を及ぼし得る．このような負荷偏りを避けるため，キャッシュの複製や分割によって個々のクライアントの負担を分散する必要がある．

2.2.4 プライバシーの保護

キャッシュを他のクライアントと共有する場合，それによるプライバシーの漏洩に注意する必要がある．しかし，プライバシーがどのような形態でコンテンツに含まれるかは多岐に渡るため，コンテンツにプライバシーを含むか否かの判別を行うことは容易ではない．また，プライバシーに関わる情報を取得する際にパラメータや Cookie によって個人を特定することが多いため，リクエストの内容についても取り扱いに注意しなければならない．

3. 関 連 研 究

3.1 キャッシュサーバを用いた負荷分散手法

この手法は，クライアントからのリクエストを複数のキャッシュサーバに振り分けることで一台当たりの負荷を軽減する．あらかじめ用意しておいたサーバ資源の量によって処理性能が決まるため，負荷を予測してサーバ資源を増強する必要がある．そのため予測が困難である Flash Crowds の対策には適しておらず，また， Flash Crowds を想定して用意したサーバ資源の大部分は平常時に余分な電力消費や運用コストを発生するといった問題もある．

Squid[9]， Backslash[10] はプロキシサーバとして動作し，クライアントからのリクエストをオリジンサーバに代わって処理する．これらを複数台用意することでオリジンサーバの負荷を軽減する．また，コンテンツ配信網 (CDN) もこの手法のひとつである．商用 CDN 最大手の Akamai[11] では，世界各地に配備したプロキシサーバを専用の高速回線を用いて接続して用いる． Akamai では莫大な量のサーバ資源を用意しているが，その量で処理性能が頭打ちになるという本質は変わらない．実際，リクエストの過剰な集中によって Akamai が利用不能になったことが過去にあり，想定を超えた負荷に耐性の無いことがわかる[12]．CoralCDN[7]，DOH[13] は専用回線の代わりにオーバーレイネットワークを用いた CDN である．これらについても Akamai と同様に資源の用意に関する問題がある．

このようにサーバ資源の増強による負荷分散は，長期的な負荷の増加への対処には適しているが Flash Crowds のように予測の難しいものへの対処としては適していない．さらに，この手法ではサービスの提供者による対策の実施が必要であるため，マッシュアップには対応できない．

3.2 クライアント間におけるキャッシュ共有による負荷分散手法

この手法は，クライアント間においてコンテンツのキャッシュを共有することでオリジンサーバの負荷を軽減する．アクセスが集中するほど利用可能なクライアント資源も増加するため， Flash Crowds のように必要な資源量の見積もりが困難な事象に適している．この手法ではコンテンツを共有可能な形に加工し管理することが必要であるが，それをサーバ側とクライアント側のどちらで行うかによって大きく二分できる．前者は，サービス提供者による対応を必要とするが，後者はサービス提供者による対応を一切必要としない．

BuddyWeb[14], BitTorrent[15], Avalanche[16], Flashback[17] はサーバ側において管理する手法である． BuddyWeb では，各クライアントが保持するローカルキャッシュを互いに提供し，どのクライアントがどのキャッシュを保持しているかを専用のサーバが管理する．他の 3 手法では，サーバは各クライアントに対してコンテンツの断片を配信し，各クライアントは断片を交換し合いながら収集し目的のコンテンツを得る．これによりサーバが各クライアントに直接コンテンツを丸ごと配信する必要が無くなるため， 1 クライアント当たりの配信負荷を大きく軽減することができる．しかしこれらの手法は，コンテンツの断片化や管理をコンテンツ提供者が行う必要があるため，マッシュアップには対応できない．

Squirrel[18] や Linga らによる手法[19] はクライアント側において管理する手法である．Squirrel は同一 LAN 内の各クライアントがローカルキャッシュを互いに提供し合うもので，これは WAN を介さないことによるコンテンツ取得の高速を目的としており Flash Crowds のような極端なアクセス集中を想定していない．また，Linga らによる手法も各クライアントが取得したコンテンツを共有するが *churn* と呼ばれるクライアントの参加・離脱の激しい状況でのキャッシュヒット率の向上などを目的としており，各クライアントの負荷やキャッシュの鮮度，またプライバシーなどの考慮も特にしていない．

4. MashCache

本章では，本論文における提案手法である MashCache の設計について述べる． MashCache はクライアント間で形成する P2P ネットワークでキャッシュを共有することによりサーバの負荷を軽減する手法で，サービス提供者による対応を必要としないためマッシュアップへの対応も可能である．MashCache は 4 で述べた要件を全て備える．すなわち， Flash Crowds の発生時期や規模の予測が不要，サービス提供者による対応が不要，クライアント間の負荷の偏りを軽減する，他者に読まれるべきでない情報を隠すことができる，といった性質を備える．

図 1 に MashCache の概要を示す．各クライアントはあらかじめキャッシュを共有するためのオーバーレイネットワークを形成しておく．そして，各クライアントはコンテンツの取得に際し，本来はサーバに対して発行するクエリを用いてこの共有ネットワークからキャッシュの取得を試みる．キャッシュヒット時はリクエストの処理は終了し，キャッシュミス時は通常通りサーバにリクエストを発行してコンテンツを取得した上でキャッシュとして共有ネットワークに提供する．そしてこれは次の 5 つの特徴によって実現されている．

- *Pure P2P Structure* 管理用サーバ等を必要とせず，負荷の集中を避ける．
- *Aggressive Caching* Flash Crowds の対象となるコンテンツの予測を不要にする．
- *Query Origin Key* コンテンツの提供元に因らないキャッシュの共有を実現する．
- *Cache Meta Data* キャッシュを分割・複製する等して扱うことを可能にする．
- *Two-phase Delta Consistency* サーバの負荷を抑えつつ高頻度なキャッシュの更新を実現する．

4.1 特　　徴

4.1.1 Pure P2P Structure

MashCache は pure-P2P 型ネットワークでキャッシュとクライアントの管理を行う．Flash Crowds の発生時にはクライアント数増加に伴って利用可能な資源が増加するため， Flash Crowds の発生時期や規模の予測が不要となる．また，キャッシュやクライアントの管理を行うためのサーバを設けないことで不要に負荷の集中点が生じることも避ける．

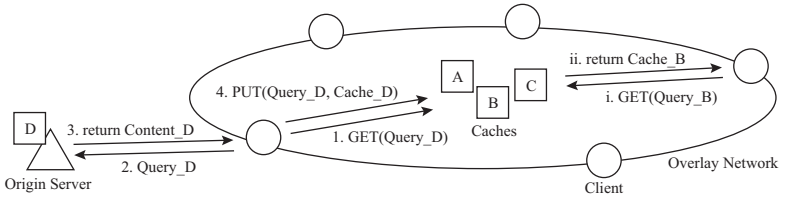

図 1　MashCache の概要

4.1.2 Aggressive Caching

MashCache は各クライアントがサーバから取得したコンテンツを全てキャッシュし，短時間で破棄するというポリシーを採る． 2.1.1 で述べたように， Flash Crowds によるアクセス数が一瞬で急増することは極めて稀であるため，Flash Crowds の対象となるコンテンツのキャッシュが常に利用可能となっていることが保証できる．また，Flash Crowds がいつ・どのコンテンツに対して生じるかを予測する必要も無い．そして，2.1.3 で述べたように，リクエストの大多数は限られたコンテンツに対するものであるため，短時間でキャッシュを破棄することで負荷低減の効果の薄いキャッシュが大量に共有されクライアント資源を逼迫することを防ぐ．

4.1.3 Query Origin Key

MashCache はキャッシュを管理するためのキーとして，通常はサーバに対して発行されるクエリを用いる．ここでクエリとはコンテンツ取得のための HTTP リクエストに含まれる URI, GET/POST メソッドのパラメータ, Cookie とする．これらはコンテンツの取得を試みるクライアントにとって既知であるため，これをキーとすることでキーとキャッシュを結びつける仕組みを別途用意する必要が無くなる．ただしクエリはプライバシーを含むことがあるため， MashCache ではクエリのハッシュをキーに用いる．これにより特定のユーザしか発行し得ないクエリを他のユーザが用いてプライバシーを含むキャッシュを取得することも防止できる．

4.1.4 Cache Meta Data

MashCache では Cache Meta Data (CMD) を導入し，キャッシュを CMD と *Chunk* の組み合わせとして扱うことで柔軟なキャッシュ管理を実現する． Chunk は各キャッシュを複製/分割する等して生成された複数のデータオブジェクトで，CMD は各 Chunk を探索

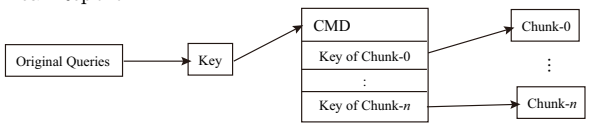

図 2 CMD を介し単一クエリから複数の Chunk を探索する方法

するためのキーを含む．こうすることで，各クライアントが自身の発行する単一クエリから複数の Chunk にアクセスすることが可能となるため，2.2.3 で述べたように各クライアントの負担を分散することができる．図 2 に，単一のクエリから目的のキャッシュの Chunk に到達するまでの様子を示す．まず，クライアントは目的のコンテンツを得るためのクエリのハッシュをキーとして CMD を探索する．次に CMD に含まれる各 Chunk のキーを用いて必要な Chunk を全て集める．取得した Chunk が分割等の加工を施されたものであれば元の状態に復元することで目的のキャッシュを得ることができる． CMD や Chunk の収集に失敗した際はキャッシュミスと見なす．CDM を介することでキャッシュ取得に要するオーバーヘッドは確実に増加するが，高画質動画のような極端にデータサイズの大きいコンテンツ等もまとめて取り扱うことが可能になる．また， CMD には Chunk を探索するためのキー以外の情報も含むことができるため MashCache に拡張性を持たせることができる．例えば Chunk のダイジェストや任意の加工を施された Chunk を復元するための情報を含むことで， Chunk の改ざんのチェックや暗号化された Chunk の復号といったことも可能となる．

4.1.5 Two-phase Delta Consistency

サーバの負荷を抑えつつも高頻度なキャッシュ更新を実現するため， CMD/Chunk について 2 つの有効期限を設定に用いる δ, Δ を定義する ($\delta < \Delta$)．各クライアントはこれらを基準に自身が管理する CMD/Chunk を他のクライアントに提供するか否かを判断する (詳細は 5.2 で述べる)．これにより， MashCache は短い間隔でキャッシュの更新をしつつサーバへの負荷を必要最小限に抑える．

5. 実　　装

MashCache の有用性を示すために Overlay Weaver 0.9.9[5] 上にプロトタイプを実装した． Overlay Weaver はオーバーレイネットワーク構築ツールキットで，多数のノードが参加するネットワークシミュレーションも行うことが可能である．

5.1 アーキテクチャ

MashCache の全機能はクライアント上に実装しており，運用する場合はブラウザプラグインやローカルプロキシとすることを想定している．サービス提供者による対応は一切不要である．図 3 に実装したプロトタイプのアーキテクチャを示す． *Proxy* はウェブブラウザ等のユーザアプリケーションからのリクエストを受け， MashCache によって得たキャッシュを返す． *Core* は以下に述べる各コンポーネントを操作する．分散ハッシュ表 (*DHT*) は共有ネットワークを構築し，これにより CMD 及び Chunk をどのクライアントが保持しているかを管理する． DHT のアルゴリズムには ChordčiteStoica2001 を採用し，Overlay Weaver に含まれる実装を利用した． Chord ではクライアント数 n に対し $O(\log n)$ のホップ数でキーに対応するクライアントを探索可能である， n が大きな値を取る場合でも探索のコストが爆発しないため， Flash Crowds 対策である本手法に適したアルゴリズムである．*Messaging* では各クライアントが保持する CMD や Chunk の授受を行う． *Downloader* はキャッシュミス時にサーバから直接コンテンツを取得する． *Key Generator* はクエリや Chunk の内容を元にして CMD や Chunk を一意に定めるキーを生成する．この実装では代表的なハッシュアルゴリズムのひとつである MD5[20] を用いた． *Data Processor* は 2 つの機能を提供する．1 つはサーバから取得したコンテンツから CMD 及び Chunk を生成すること，もう 1 つは CMD と Chunk から元のコンテンツを復元することである．ここでの処理としては，分割/結合，圧縮/解凍，暗号化/復号化，改ざん検知等を想定しているが，今回は最も単純な例として分割/結合の機能を実装した． *Data Manager* はローカルに保持している CMD と Chunk の有効期限を管理しており，必要に応じてデータの更新や削除等を行う．

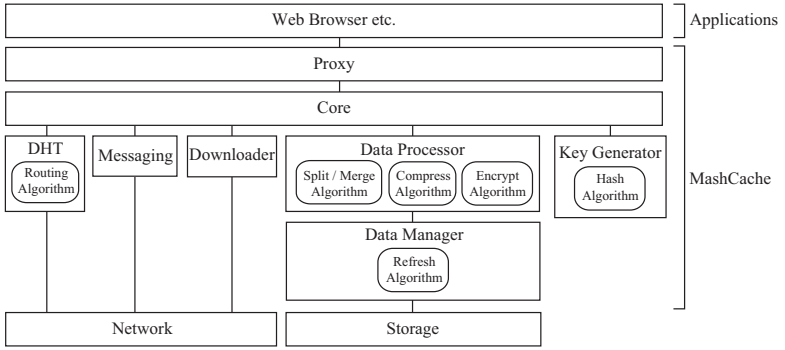

図 3 MashCache のアーキテクチャ

5.2 Procedures

MashCache を利用するクライアントは DHT によりあらかじめ共有ネットワークに参加

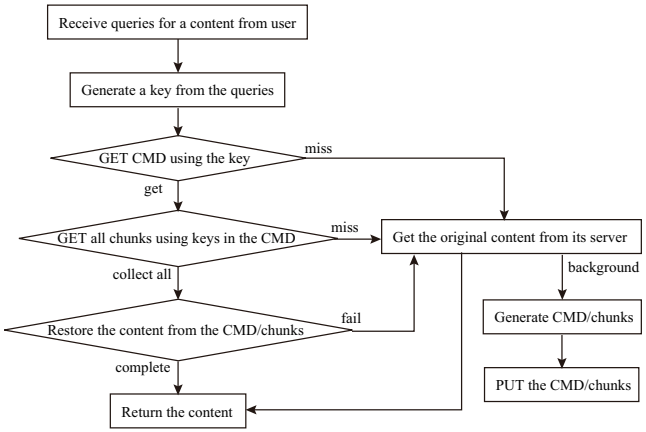

図 4 リクエスト処理の流れ

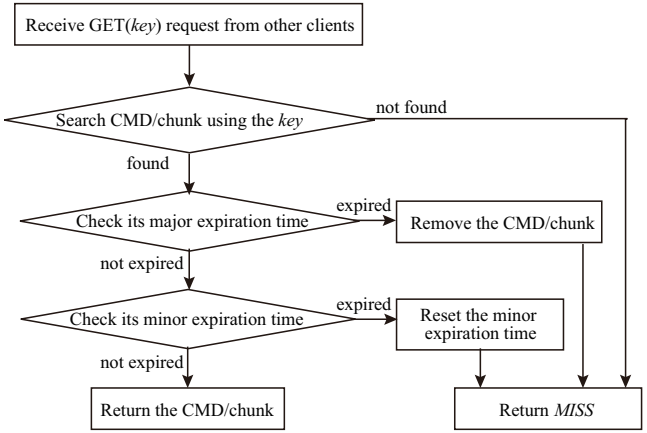

図 5 CMD/Chunk 取得の流れ

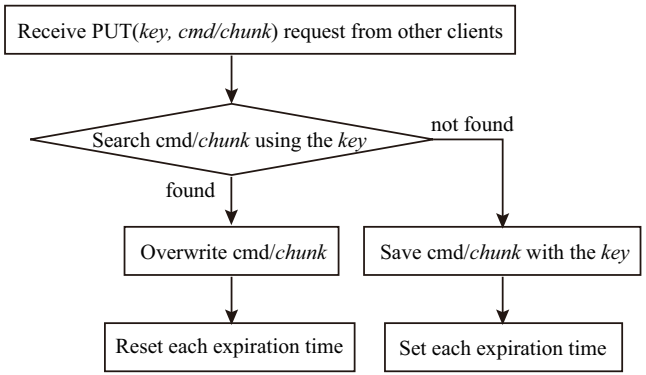

図 6 CMD/Chunk 配置の流れ

しておく必要がある．共有ネットワークへの参加方法は DHT のアルゴリズムに因るが，本実装では既に共有ネットワークへ参加している任意のクライアントをブートストラップとして Join 処理を行う．このネットワークでは Key Generator が生成したキーを用いて CMD 及び Chunk を管理する．

図 4 に，クライアントのリクエストが MashCache 内で処理されてコンテンツが返されるまでの流れを示す．各クライアントが発行したリクエストは Proxy にて受理される．クライアントはリクエストに含まれるクエリのハッシュを求め，これをキーとして CMD を探索する． CMD を取得できた場合は中に含まれるキーを用いて必要な Chunk を探索・取得する．ここで CMD や Chunk の要求を受けたクライアントは 5.2.1 に従ってデータを渡す．CMD や Chunk の取得や元のキャッシュの復元に失敗した場合はキャッシュミスとし，改めてサーバにコンテンツを直接リクエストする．サーバからコンテンツを取得したクライアントはそれを基に CMD と Chunk を生成し，それぞれのキーに対応するクライアントに送信する．対応するクライアントは 5.2.2 に従って受け取った CMD や Chunk を処理する．5.2.1, 5.2.2 において，2 の有効期限を用いてキャッシュの管理をする仕組みが 4.1.5 で触れた Two-phase Delta Consistency である．

5.2.1 CMD/Chunk の取得

CMD/Chunk を取得する際はその *key* に対応するクライアントを共有ネットワークで探索し GET(*key*) メッセージを送信する． GET(*key*) メッセージを受信したクライアントは自身の Data Manager にて対象キャッシュの有無及び有効か否かを確認し，それに応じた対応を取る．図 5 にその処理の流れを示す．キャッシュが配置された時刻を t_0， *minor expiration time*， *major expiration time* をそれぞれ $(t_0+\delta)$, $(t_0+\Delta)$ とする $(\delta < \Delta)$. ここで GET(*key*) メッセージを受けた時刻を t とすると，$(t_0+\Delta) < t$ の場合はキャッシュを破棄，$(t_0+\delta) < t < (t_0+\Delta)$ の場合はキャッシュミス扱いにした上で minor expiration time を $(t+\delta)$ に再設定する．これにより，キャッシュを無効とせずに更新処理を開始することが可能になる． Flash Crowds 発生時にキャッシュが無効になると，次に利用可能になるまでの間のアクセスがサーバに集中してしまう．しかし本手法ではキャッシュミスを起こしたクライアントのみがサーバへアクセスし他のクライアントは引き続き有効になっているキャッシュを利用することができるため，Flash Crowds 時にも必要最低限のサーバアクセスでキャッシュの更新処理を行うことが可能となる．

5.2.2 CMD/Chunk の配置

CMD/Chunk を配置する際はをその *key* に対応するクライアントを共有ネットワークで探索し， PUT(*key, cmd/chunk*) メッセージを送信する．PUT メッセージを受けたクライアントは自信の Data Manager を確認し，*key* をキーとするエントリが存在すれば *cmd/chunk* で上書きし，存在しなかったらこのペアを新たに追加する．いずれの場合も minor/major expiration time は δ, Δ を用いて改めて設定する．図 6 にその処理の流れを示す．

6. 評　　価

本章では MashCache の実装を用いて評価実験を行うことで，提案手法が Flash Crowds

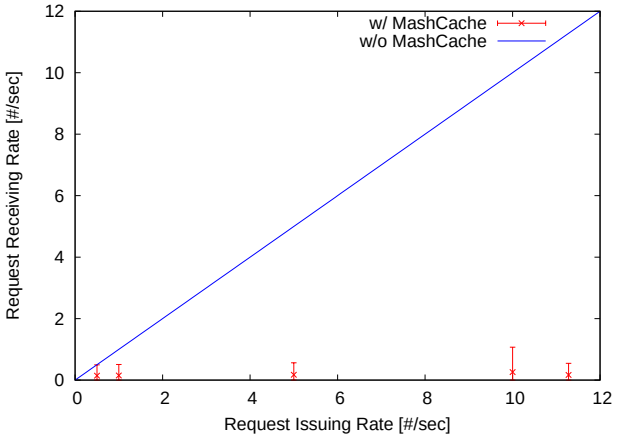

図 7 リクエストの発行頻度と処理頻度

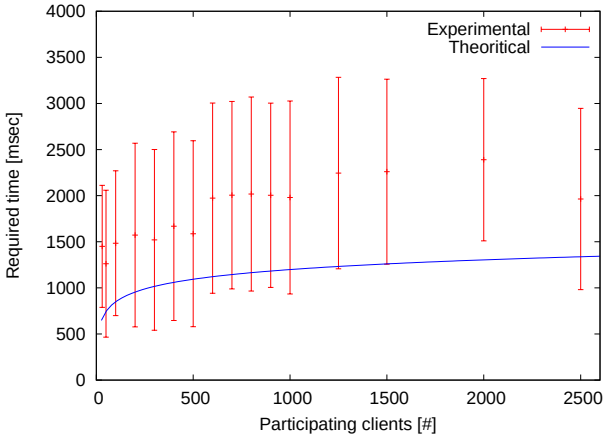

図 8 キャッシュ取得に要した時間

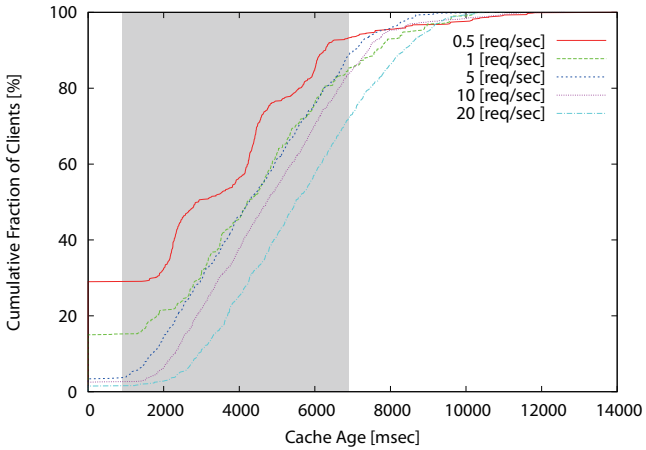

図 9 キャッシュの鮮度

対策として有用であることを示す．実験にはシミュレータ用とコンテンツサーバ用の 2 台の物理マシンを用意し，最大で 2,500 台の仮想クライアントによるコンテンツサーバへのアクセス集中をシミュレートした．シミュレータは 3.00 GHz クアッドコア CPU (Intel Xeon), RAM 12 GB，Gigabit Ethernet 規格の NIC を搭載し，Linux-2.6.20 が稼働していた．コンテンツサーバは 2.40 GHz デュアルコア CPU (Intel Core2Duo), RAM 2 GB，Gigabit Ethernet 規格の NIC を搭載し，Linux-2.6.31 と Apache HTTP サーバ 2.2.12[21] が稼働していた．また，仮想クライアント間の通信遅延には *Meridian* データセットを用いた．これは Meridian project[22] にて計測された 2,500 台の異なる DNS サーバ間の通信遅延である．さらに各仮想クライアントの通信速度は 10 Mbps に制限した．Chunk サイズの上限は 256 KB とし，δ は 5 秒，Δ は 10 秒に設定した．コンテンツデータは 1 MB のランダムバイト列を用いた．

6.1 Mitigating Request Rates

MashCache がウェブサーバが処理するリクエスト数を減少させることを確認するため，クライアントによるリクエスト発行数とサーバにおけるリクエスト処理数を調べた．図 7 はクライアントにおける一秒間当たりのリクエスト発行数とコンテンツサーバにおける一秒間当たりのリクエスト処理数との関係を示す．ここで “w/ MashCache” の各プロットは平均値でエラーバーは標準誤差を示す． MashCache を用いない場合発行されたリクエストは全てサーバで処理されるが，MashCache を用いた場合にサーバが処理したリクエスト数は発行数に因らずおよそ毎秒 0.2 件のであった．これは minor expiration time を過ぎたクライアントによるサーバへのリクエスト発行がおよそ 5 秒に 1 件の割合で生じたためだと考えられる．

6.2 Scalability and Overhead

クライアント数が増加にしても実用性を保つことを確認するために，クライアント数とキャッシュ取得に要する時間の関係を調べた．図 8 は共有ネットワークへ参加しているクライアント数とリクエストを発行してから CMD 及び Chunk を収集しキャッシュの復元を完了するまでに要した時間の関係を示す．理論値を示す曲線は以下に記述する計算式によって求められる $\bar{t}_{\mathrm{cache}}$ を描画したものである．理論値の計算には通信遅延やデータ転送時間のみ用いており，各クライアントのローカル処理 (e.g., ハッシュ値の計算，キャッシュの復元) は考慮していない．また，実験値のプロットは，クライアントの数をある値にした際にかかった所要時間の平均値で，エラーバーは標準誤差を示す．

理論値の算出にあたって次の変数を定義する． n: クライアント数，$\bar{l}$: クライアント間通信遅延の平均値， v: 各クライアントのネットワーク帯域， V_{foo}: *foo* のデータサイズ，$\bar{t}_{\mathrm{search}}$: あるキーに対応するクライアントの探索に要する時間の平均値，$\bar{t}_{\mathrm{cmd}}/\bar{t}_{\mathrm{Chunk}}/\bar{t}_{\mathrm{cache}}$: CMD/Chunk/キャッシュの取得に要する時間の平均値， $\bar{t}_{\mathrm{content}}$: ウェブサーバからのコンテンツ取得に要する時間の平均値．

$$\bar{t}_{\text{search}} = \bar{l} \log n \tag{1}$$

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

$$\bar{t}_{\text{chunk}} = \bar{t}_{\text{search}} + \bar{l} + \frac{V_{\text{chunk}}}{v} \tag{3}$$

$$\bar{t}_{\text{cache}} = \bar{t}_{\text{cmd}} + \bar{t}_{\text{chunk}} \left\lceil \frac{V_{\text{content}}}{V_{chunk}} \right\rceil \tag{4}$$

$$\bar{t}_{\text{content}} = \bar{l} + \frac{V_{\text{content}}}{v} \tag{5}$$

キャッシュミス時は，ミスが判明するまでに行った処理時間にウェブサーバからコンテンツの取得を行う時間が加わる．従って最悪の場合は，必要な CMD 及び Chunk を全て収集した上でキャッシュの復元に失敗するケースで，所要時間は $\bar{t}_{\text{cache}} + \bar{t}_{\text{content}}$ となる．

6.3 Cache Freshness

MashCache が各クライアントに対して新しいキャッシュを提供できていることを確認するために，各クライアントが取得したキャッシュの経過時間を調べた．図 9 は各クライアントが取得したキャッシュの経過時間とそれよりも新しいキャッシュを取得したクライアントの割合の関係を示す．但し，経過時間は CMD が生成された時刻から計測し，経過時間 0 秒はコンテンツサーバから直接取得したことを示すものとする．多くの場合，CMD 及び Chunk の配置が完了してから更新されるまでに取得されるので $\bar{t}_{\text{cache}}$ から $\bar{t}_{\text{content}} + 2\bar{t}_{\text{cache}} + \delta$ の間のキャッシュを得ることになる．前者はキャッシュの配置に要する時間，後者は minor expiration time までの時間とサーバからコンテンツを取得する時間とキャッシュの再配置に要する時間を加えたものになる．本実験におけるパラメータを用いて計算したものが図 9 のグレーの範囲である．また，最悪の場合は major expiration time を過ぎる直前になるため $\bar{t}_{\text{cache}} + \Delta$ となる．

6.4 Load of Cache Updating

6.3 に示したように MashCache は頻繁なキャッシュの更新を行うが，ここではそれに伴うウェブサーバへの負荷を低く抑えていることを示す．図 10, 11 は共に，毎秒のリクエスト発行数を 10 件とした場合にサーバにおけるリクエスト処理数がどのように推移したかを示す．キャッシュの期限に Δ のみ用いる場合が図 10 で， Two-phase Delta Consistency を採用したものが図 11 である．前者はキャッシュの失効から再配置完了までの時間に発行されたリクエストがサーバに集中しているが，後者はキャッシュの更新を担当することになったクライアントのリクエストのみがサーバに到達している．このように MashCache では高頻度なキャッシュ更新を行う際にもサーバの負荷を低く抑えることが可能であり，オリジナルのコンテンツの更新がキャッシュに反映されないという状況を少なくする．実運用ではキャッシュの更新に失敗する可能性も考慮し，Δ を δ よりも何倍も大きくしてキャッシュ失効前に更新する機会を何度か確保することが望ましい．

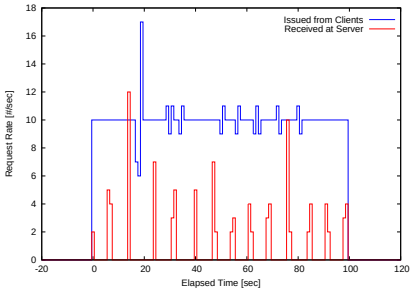

図 10 キャッシュミスによるサーバ負荷の推移 1

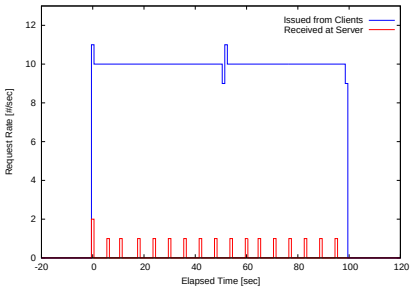

図 11 キャッシュミスによるサーバ負荷の推移 2

7. おわりに

本論文では，マッシュアップにも対応可能な Flash Crowds 対策として *MashCache* を提案した． MashCache ではクライアント間で形成する Pure-P2P 型ネットワークでキャッシュの管理を行うことで，サービス提供者側での対応を必要とせずにキャッシュを共有することが可能である．本研究は Flash Crowds の詳細な分析に基づいており，ピークを迎えるまでに時間的余裕がある，ピークの継続は数時間程度である，リクエストの大部分が限られたコンテンツに集中するといった特性に着目し，あらゆるコンテンツのキャッシュを生成し，人気の無いコンテンツのキャッシュはすぐに消す仕組みを取り入れるなど Flash Crowds に適した設計になっている．また，クライアント間の負担の偏りの是正や高頻度かつ低負荷なキャッシュの更新，また，利用者のプライバシーを守る仕組みも備える．プロトタイプの実装を用いたシミュレーションでは，これらの機能が有効に働き，クライアント間の連携のみでサーバの負荷を低減することが確認できた．

謝辞 本研究の一部は，文部科学省科学研究費補助金特定領域研究「情報爆発時代に向けた新しい IT 基盤技術の研究」による支援を受けている．

参 考 文 献

1) Amazon.com, Inc.: Amazon. `http://www.amazon.com/`.
2) Kembel, R.W.: Fibre Channel: A Comprehensive Introduction, p.8 (2000).
3) Jung, J., Krishnamurthy, B. and Rabinovich, M.: Flash Crowds and Denial of Ser-

vice Attacks: Characterization and Implications for CDNs and Web Sites, *Proc. of Int'l Conference on WWW* (2002).
4) Freedman, M. J.: Experiences with CoralCDN: A Five-Year Operational View, *Proc. of USENIX Symposium on Networked Systems Design and Implementation* (2010).
5) 首藤一幸，田中良夫，関口智嗣：オーバレイ構築ツールキット Overlay Weaver，情報処理学会論文誌：コンピューティングシステム， Vol.47, No.SIG12 (ACS 15), pp. 358–367 (2006).
6) SourceForge, Inc.: Slashdot. `http://slashdot.org/`.
7) Freedman, M.J., Freudenthal, E. and Maziéres, D.: Democratizing content publication with Coral (2004).
8) Amazon Web Services LLC: Amazon Elastic Compute Cloud. `http://aws.amazon.com/ec2/`.
9) Wessels, D., Nordstrom, H., Rousskov, A., Chadd, A., Collins, R., Serassio, G., Wilton, S. and Francesco, C.: Squid: Optimising Web Delivery. `http://www.squid-cache.org/`.
10) Stading, T., Maniatis, P. and Baker, M.: Peer-to-Peer Caching Schemes to Address Flash Crowds, *Proc. 1st International Workshop on Peer-to-Peer Systems* (2002).
11) Akamai Technologies: Akamai. `http://www.akamai.com/`.
12) Akamai Technologies: Akamai Provides Insight into Internet Denial of Service Attack. `http://www.akamai.com/html/about/press/releases/2004/press_061604.html`.
13) Jernberg, J., Vlassov, V., Ghodsi, A. and Haridi, S.: DOH: A Content Delivery Peer-to-Peer Network, *Proc. of European Conference on Parallel Computing* (2006).
14) Wang, X., Ng, W., Ooi, B., Tan, K.-L. and Zhou, A.: BuddyWeb: a P2P-based Collaborative Web Caching System, *Proc. International Workshop on Peer-to-Peer Computing* (2002).
15) Cohen, B.: Incentives Build Robustness in BitTorrent, *Proc. of Workshop on Economics of Peer-to-Peer Systems* (2003).
16) Gkantsidis, C., Miller, J. and Rodriguez, P.: Comprehensive view of a live network coding P2P system, *Proc. of ACM Special Interest Group on Data Communications Conference* (2006).
17) Deshpande, M., Amit, A., Chang, M., Venkatasubramanian, N. and Mehrotra, S.: Flashback: A Peer-to-Peer Web Server for Flash Crowds, *Proc. of IEEE International Conference on Distributed Computing Systems* (2007).
18) Iyer, S., Rowstron, A. and Druschel, P.: Squirrel: A decentralized peer-to-peer web cache, *Proc. of ACM Symposium on Principles of Distributed Computing* (2002).
19) Linga, P., Gupta, I. and Birman, K.: A Churn-Resistant Peer-to-Peer Web Caching System, *Proc. of ACM Workshop on Survivable and Self-Regenerative Systems* (2003).
20) Rivest, R.: The MD5 Message-Digest Algorithm, *RFC1321* (1992).
21) The Apache Software Foundation: Apache HTTP Server. `http://httpd.apache.org/`.
22) Wong, B., Slivkins, A. and Sirer, E.G.: Meridian: A Lightweight Network Location Service without Virtual Coordinates, *Proc. of ACM Special Interest Group on Data Communication*, pp.85–96 (2005).