学位論文　博士（工学）

需要変動によるサーバ負荷の変動を軽減するコンテンツ配信手法に関する研究

2015 年度

慶應義塾大学大学院理工学研究科

堀江 光

需要変動によるサーバ負荷の変動を軽減するコンテンツ配信手法に関する研究

堀江 光

論文要旨

インターネット利用者数の増加に伴い，多くのウェブサービスが運用され，その経済的および社会的重要性が増している．しかし，その一方で，Flash Crowds と呼ばれる現象がウェブサービスの安定的な運用を妨げるとして問題となっている．Flash Crowds とは，特定のサービスやコンテンツに対する突発的なアクセス集中のことで，数分の間にウェブサービスへのアクセス数が平常時の数百倍から数千倍にも達するとの報告がある．Flash Crowds によりサーバが過負荷状態となるとサービスの性能低下や異常停止などの障害の原因となるため，対策が必要である．

Flash Crowds による過負荷状態を防ぐためには，キャッシュ配信によりサーバへ到達するリクエスト数を減少させる負荷軽減と，サーバを複製することで個々のサーバへの負荷を分散する負荷分散の両方が必要である．なぜならば，ウェブサービスが配信するコンテンツには静的コンテンツと動的コンテンツの 2 種類が存在し，静的コンテンツには負荷軽減手法，動的コンテンツには負荷分散手法が適するためである．また，Flash Crowds に対応するためには，各手法において 4 つの性質を備える必要がある．第一に，投入した資源量に対し十分な処理能力を得られること，第二に，Flash Crowds 発生後速やかに処理能力が向上すること，第三に，需要に対して過不足のない適切な資源量で稼働すること，第四に，コンテンツの更新内容が速やかに反映されることである．既存手法は プロキシ方式，キャッシュ共有方式，クラウド方式，完全複製方式 の 4 種に大別できるが，これら 4 つの性質をすべて満たすものはない．

本論文では，Flash Crowds のような突発的な需要変動によるサーバ負荷の変動を軽減するコンテンツ配信手法を提案する．本提案は，クライアント間連携による負荷軽減手法 *MashCache* およびデータセンタ間連携によるストレージの負荷軽減手法 *Pangaea* からなり，それぞれ 4 つの性質を満たす．

MashCache は，各クライアントが取得したコンテンツをキャッシュとして他のクライアントと共有することで，ウェブサービスに直接アクセスすることなく目的とするコンテンツの取得を可能とする．キャッシュの共有には分散ハッシュ表によ

る Peer-to-Peer ネットワークを用いる．また，MashCache は，Aggressive Caching, Query Origin Key, Cache Meta Data, Two-phase Delta Consistency の 4 つの要素技術により，各クライアントやウェブサービスの負荷を低く保ちつつ，常に新しいキャッシュをクライアント間で共有することを可能とする．

Pangaea は，複数の異なるデータセンタに属するストレージサーバを統合し単一のキーバリューストア (KVS) を構築することで，個別のデータセンタ規模の制約を受けない負荷分散を実現する．一方で，データセンタ間通信路は高遅延かつ狭帯域であるため，応答時間や転送速度の性能低下が問題となる．Pangaea は Multi-Layered Distributed Hash Table, Local-first Data Rebuilding の 2 つの要素技術により，データセンタ間を跨ぐ通信の頻度と転送量を低下させる．これにより，複数のデータセンタの資源の効果的な利用を実現する．

提案手法の Flash Crowds に対する有用性を確認するために，シミュレータおよび実際のインターネット環境を用い評価した．MashCache がサーバの負荷を軽減していることを確認するため，2,500 台のクライアントによる Flash Crowds を模した負荷を発生させたところ，ウェブサーバの負荷が約 98.2 % 削減された．また，コンテンツの更新が速やかに反映されることを確認するため，各リクエストによって得られたコンテンツのキャッシュがいつ生成されたものであるかを解析したところ，95 % 以上のリクエストが過去 10 s 以内に生成されたキャッシュで処理された．また，Pangaea がデータセンタ間通信による性能低下を軽減し，負荷分散手法として有用であることを確認するために，シミュレータと実際のインターネット環境を用いて評価したところ，Pangaea によりデータの探索に要する平均時間が 74 %，データセンタ間のデータ転送量が 70 % 削減された．

A Study on Stable Content Distribution Dealing with Demand Fluctuation

Hikaru HORIE

Abstract

As the number of Internet users continue to increase, many web services running on the Internet gain economic and social importance. As the Internet population increases, *flash crowds* often occur on the Internet and prevent web services from running stably. A huge number of clients suddenly flock to a web service in flash crowds, and the number of requests increases hundred to thousand times as many as usual in a few minutes. It is necessary to take measures against flash crowds because server overload results in performance degradation or service outage.

To keep web services stable, it is necessary to reduce the burden on each server by both *load-reduction* and *load-balancing*. The former reduces the number of requests that reach the servers by deployment of caches. The latter distributes requests that reach the servers by server replication. The reason why both of them are necessary is that load-reduction and load balancing are suitable for static contents and dynamic contents respectively. To deal with flash crowds, load-reduction techniques and load-balancing techniques must satisfy four requirements: scalability, agility, elasticity and consistency. Existing techniques can be classified broadly into four categories: proxy-based, cache-sharing, cloud-based and full-replication. Unfortunately, there is no technique that satisfies the all four requirements.

This dissertation proposes techniques to deal with demand fluctuation such as flash crowds for stable web services. The proposed techniques consist of *MashCache*, a load-reduction technique, and *Pangaea*, a load-balancing technique.

MashCache reduces the number of requests that reach web services by sharing caches between clients. MashCache uses a fully-decentralized peer-to-peer network built with a distributed hash table to manage caches and clients. MashCache enables us to keep caches updated and the burden on web servers and clients low by four elemental technologies: Aggressive Caching, Query Origin Key, Cache Meta Data and Two-phase Delta Consistency.

Pangaea distributes the burden on storage servers regardless of which datacenter they are placed. Pangaea unifies storage servers in multiple datacenters as a single uniform key-space key-value store and enables us to exceed the limited capacity of a single datacenter. The key-value store enables service providers to flexibly deploy web servers or application servers to any datacenter they want to use. However, inter-datacenter communication channels are high-latency and narrow-bandwidth and degrade the performance of the key-value store. Pangaea reduces opportunities and amount of inter-datacenter communications by two elemental technologies: Multi-Layered Distributed Hash Table and Local-first Data Rebuilding.

Evaluation results with simulation and real Internet environments demonstrate the proposed techniques are suitable for flash crowds. MashCache reduces the number of the requests that 2,500 clients issue and that reache a web server by about 98.2 %. 95th percentiles of the requests are processed with caches which are generated in the past 10 seconds. Using two datacenters and 500 servers in each datacenter, Pangaea reduces the average time to find a data object by 74 % and the amount of inter-datacenter data transfer by 70 %.

目 次

図 目 次

表 目 次

第1章　序論

1.1　本研究の背景と動機

近年，インターネット (The Internet) の利用者数は増加の一途を辿っており，その社会的重要性も増している．インターネットとは，インターネットプロトコル [1] を用いて世界中のネットワークを相互に接続したものである．当初は主に学術研究を目的として運用されていたが，近年では商用利用が 97 % を占め，社会的に不可欠な存在となっている [2]. Internet World Stats によるとインターネット利用者数は増加を続けており，2014 年現在で約 30 億人が利用している [3]．更に 2000 年から 2014 年までの増加率は 741 % であり，今後も更なる増加が見込まれている．このような利用者数の増加に伴い，ウェブサービスの数や重要性も高まっている．例えば，ショッピングサイトにおける応答遅延が 100 ms 増加すると売上が 1 % 減少するとの報告 [4] やショッピングサイト大手の `Amazon.com` [5] のサービスが 1 時間停止すると約 $ 180,000 の損害が発生するとの試算 [6, 7] がある. 更に，近年では災害発生時の情報伝達手段 [8–11] としても利用されており，経済面だけでなく社会基盤としての重要度を増している．

このような利用者数の増加や重要性の高まりの一方で，Flash Crowds と呼ばれる現象が問題となっている．Flash Crowds とは，特定のサービスやコンテンツに対する突発的なアクセス集中のことで，サーバ資源の許容量を超える需要が発生するとサービスの性能低下や異常停止といった障害の原因となる．Flash Crowds 自体は通常のネットワークアクセスであり，悪意を持ってサーバ負荷を増大させることで障害を引き起こすサービス拒否攻撃 (DoS 攻撃) [12, 13] とは異なるが，現象としては類似している．一般的にアクセス集中に対しては，十分なサーバ資源を用意することによる負荷分散手法が有効である．しかし，Flash Crowds は様々な要因で発生するため，その発生の時期, 規模, 対象の予測が困難である．

Flash Crowds の要因には，第三者による情報拡散や事件，事故，災害等の偶発的な事象が挙げられる．第三者による情報拡散については，更に 3 つに大別すること

ができる [14]．第一に，公開情報一覧への掲載が挙げられる．ニュースサイトやポータルサイト等の多数のユーザが利用するサイトに紹介が掲載されると，その紹介を経由して多数のユーザが掲載サービスへアクセスをする．特に Slashdot [15] のように知名度が高くユーザの多いサイトへの掲載の影響は大きく，Slashdot Effect [16] などと呼ばれるほどその影響は大きい．第二に，放送が挙げられる．テレビやラジオによる放送は同時に多数のユーザに情報を届けるため，それを視聴して行動を起こしたユーザによるアクセスは短期間に集中する傾向にある [17]．第三に，個人間情報伝播が挙げられる．例えば，Facebook [18] 等の Social Networking Service (SNS) や Twitter [19] 等のコミュニケーションサービスにおける個人間のコミュニケーションを通じて情報が拡散する．これらのサービスは “シェア” や “リツイート” 等の簡単に情報拡散する機能を備えていることもあり拡散の速度が速く，短い期間に多くのユーザに伝達し，Flash Crowds を発生させる．これらのような，第三者による情報拡散をサービス運営者が把握し予測することは困難である．また，事件，事故，災害等の偶発的な事象については更に予測が困難である．例えば，大地震が発生した場合は災害伝言板 [8–11] へは安否確認のためのアクセスが集中するが，地震の発生を予測することは困難である．

Flash Crowds の対策には，その発生時期や規模の予測に基づかない手法が必要である．一般的にアクセス集中に対しては，サーバの複製やキャッシュの配信等によって，個々のサーバに対する負荷を分散することが有効である．この方法では，十分な量のサーバ資源を用意することにより，各サーバの負荷を軽減し過負荷によるサービス障害の発生を抑制することができる．一方，Flash Crowds は発生時期や規模の予測が困難であるため，サービス管理者が適切な量の資源を適切な時期に用意することも困難である．Flash Crowds に対応するため常時大量の資源を稼働することは，平常時に過剰な量の資源が稼働していることを意味し，経済的および環境的負担が大きい．Elson らは，ウェブサービスの配信に大量の資源を用いることでより大きな負荷に耐える技術 [20–28] を挙げる一方で，使用する資源の量に応じて運用コストが増大していくことを指摘している [29]．また，そのような負担を許容して備えたとしても，予測を上回る規模の負荷が発生した場合には対応することが出来ない．従って，正確な予測ができない以上，本質的な解決作ではない．

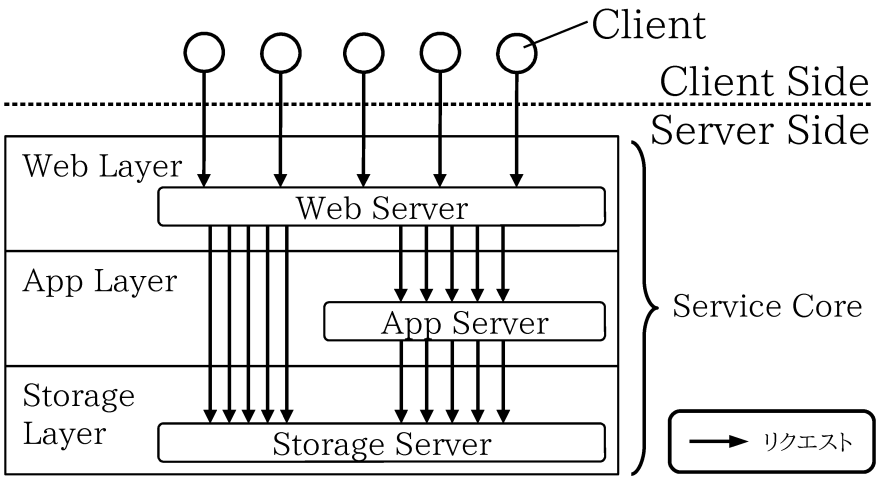


図 1.1: ウェブサービスの構成例

1.2 ウェブサービスの基本構成

本節では，本研究が対象とする，ウェブサービスの配信に関連するソフトウェアコンポーネントの一般的な構成について整理する．図 1.1 にウェブサービスを構成するソフトウェアコンポーネントの例を示す．各コンポーネントは，サービス利用者が管理主体であるクライアント側とサービス提供者が管理主体であるサーバ側に大別することができる．

1.2.1 クライアント側コンポーネント

サービス利用者は，インターネット接続が可能なパーソナルコンピュータやスマートフォン等の情報端末を用いてウェブサービスを利用する．ウェブサービスを利用するためにサービス利用者は，端末にインストールされたウェブブラウザ [30–34] やウェブサービス固有のクライアントアプリケーションなどを通じて，ウェブサービスにアクセスし必要な情報を送受信する．ウェブブラウザは，ウェブサービスのフロントエンドであるウェブサーバと通信を行う．ウェブサーバについては第 1.2.2 項にて詳細を述べる．

一般的にウェブブラウザやサービス固有のアプリケーションは，Hypertext Transfer Protocol (HTTP) [35–39] というステートレスなプロトコルによりウェブサーバと通信を行う．ウェブブラウザはウェブサーバに対し，使用するプロトコルに則ってリクエストを送信し，必要なコンテンツを取得する．

```
1 GET /item/?id=12345 HTTP/1.1
2 Host: www.shopping-site.com
3 Connection: keep-alive
4 Accept: text/html,application/xhtml+xml,application/xml;q↙
    ↳ =0.9,image/webp,*/*;q=0.8
5 User-Agent: Mozilla/5.0 (X11; Linux x86\_64) AppleWebKit↙
    ↳ /537.36 (KHTML, like Gecko) Chrome/43.0.2357.81 ↙
    ↳ Safari/537.36
6 Accept-Encoding: gzip, deflate, sdch
7 Accept-Language: ja,en-US;q=0.8,en;q=0.6
8 Cookie: session-id=d8f79394-0aa3-41b3-be9a-0e5e70886405
```

図 1.2: HTTP リクエストヘッダの例

図 1.2, 図 1.3 に HTTP におけるリクエストとレスポンスの例を示す．リクエストにおいてウェブブラウザは，図 1.2 の 1 行目のように，必要なコンテンツを指定する．リクエストを受けたウェブサーバは，その内容に応じて必要なコンテンツをレスポンスとして返す．ここで HTTP は自体はステートレスなプロトコルであるため，ひと組のリクエストおよびレスポンスで処理が完結する．しかし，多くのウェブサービスではユーザの識別や複数のリクエストを跨ぐ処理が必要となるため，一連の処理をセッションとして扱う仕組みを備える．セッションは，たとえば，ウェブサービスへのログイン状態や，ショッピングカートに入れた品物の保持などに活用される．

セッションを実現するためにウェブブラウザは，サーバから発行されたセッション識別子を保持し，必要に応じてこれをウェブサーバに送信することで一連のリクエストが同一のセッションに属することを示す．ウェブサーバは，リクエストに含まれるセッション識別子を用いて各クライアントの状態を管理する．したがって，セッション識別子は他のユーザに対して秘密でなければならない．

セッション識別子を利用するための方法は主に 1) Cookie を用いる方法 [40] と 2) リクエストパラメータを用いる方法がある．Cookie とは，ウェブブラウザとウェブサーバ間で状態を管理するための手法およびその保存データを指す．Cookie を用いる場合，ウェブブラウザはウェブサーバから発行されたセッション識別子を Cookie に保存し，それ以降のリクエストにおいてリクエストヘッダに含める．Cookie の保存や送信は図 1.2 の 8 行目, 図 1.3 の 11 行目のように行う．一方リクエストパラメータを用いる場合は，ウェブサーバがセッション識別子を入力フォームのデフォルト値に設定するなどしておき，それを次回のリクエストに含ませることでセッションを判別する．

```
1  HTTP/1.1 200 OK
2  Date: Fri, 29 May 2015 10:59:32 GMT
3  Server: Server
4  pragma: no-cache
5  cache-control: no-cache
6  x-frame-options: SAMEORIGIN
7  expires: -1
8  Vary: Accept-Encoding,User-Agent
9  Content-Encoding: gzip
10 Content-Type: text/html; charset=UTF-8
11 Set-cookie: session-id-time=92f53230-0fbd-4156-aa40↙
       ↳ -0476682a84b8; expires=Tue, 01-Jan-2036 08:00:01 GMT
```

図 1.3: HTTP レスポンスヘッダの例

1.2.2 サーバ側コンポーネント

サービス提供者は，インターネットからの接続を受け付けるサーバを運用し，クライアントに対してウェブサービスを提供する．コンポーネントの構成はサービスの内容に依存するが，ウェブ層，アプリケーション層，ストレージ層の 3 層構造での運用が代表的である．本論文では，ウェブサービスを運用するために必要なこれらのコンポーネントをまとめて，サービスコアと呼ぶ．各層では，ウェブサーバ，アプリケーションサーバ，ストレージサーバが稼働する．但し，これらは論理的な構成であり，必ず異なるマシン上で各サーバが動作することを意味するものではない．たとえば，実際の運用において，同一マシン上でウェブサーバとアプリケーションサーバが動作することがある．

ウェブ層

ウェブ層ではウェブサーバが稼働する．ウェブサーバは，クライアントからの HTTP 等による接続を受け付け，リクエストに応じたコンテンツを返す．代表的なウェブサーバには Apache [41], Nginx [42], Internet Information Service (IIS) [43] 等がある．ウェブサーバが配信するコンテンツは，静的コンテンツと動的コンテンツに大別することができる．静的コンテンツは，画像ファイルや動画ファイル等のように予めコンテンツの内容が定まっているものを指す．ウェブサーバは，ストレージサーバからリクエストに応じた静的コンテンツを取得し，クライアントに返す．一方，動的コンテンツは，クライアントからのリクエストや時刻等その他の条件により内容が変化するテキスト等，リクエストを受けた後に内容が定まるものを指す．ウェブサーバはクライアントからのリクエストをアプリケーション

サーバに転送し，アプリケーションサーバにおける処理結果をクライアントに返す．処理内容は，アプリケーションサーバで動作するアプリケーションおよびリクエスト内容等に依存する．

アプリケーション層

アプリケーション層ではアプリケーションサーバが稼働する．アプリケーションサーバは，ウェブサーバから受け取ったリクエストに応じた処理を実行し，処理結果をウェブサーバに返す．アプリケーションサーバで実行されるアプリケーションはサービスにより大きく異なるが，多くのウェブサービスに共通する機能はウェブアプリケーションフレームワークとして提供されており，各種プログラミング言語で広く利用されている [44–49]．また，各アプリケーションは Web Server Gateway Interface (WSGI) [50], Common Gateway Interface (CGI) [51] 等のインターフェースを通じてウェブサーバと通信を行う．アプリケーションサーバとウェブサーバは統合される場合もある．たとえば，Apache, Nginx はそれぞれアプリケーションサーバとしての機能をモジュールとして備えている．

ストレージ層

ストレージ層ではストレージサーバが稼働する．ストレージサーバは，ウェブサーバやアプリケーションサーバからのリクエストに応じて，コンテンツを返す．ストレージサーバには，大量のデータ保存が可能な大容量と高い I/O スループットが求められるため，専用の物理マシンを用意することが一般的である．ストレージサーバはファイルシステムとデータベース管理システムの 2 つに大別できる．

ファイルシステムは，主に静的コンテンツを保持しておくために用いられる．ストレージサーバで用いられるファイルシステムは，マシンを多数用意してクラスタとすることで大量のデータ保存や I/O の並列化によるスループット向上を実現する．このようなファイルシステムには，たとえば，Network File System (NFS) [52–54], Ceph [55], Lustre [56], Google File System (GFS) [57, 58], Hadoop Distributed File System (HDFS) [59] などがある

データベース管理システムは，動的にデータを読み書きする場合に用いられ，更に代表的なデータベース管理システムは関係データベース管理システム (RDBMS) とキーバリューストア (KVS) の 2 つに分けられる．RDBMS は関係モデル [60]

に基づき，強い一貫性の下に構造化されたデータを管理する．RDBMS では一般的に，トランザクション処理など複雑なデータ操作を実現しており，これらの操作は SQL [61] を用いて行う．このような RDBMS には，たとえば，MySQL [62], PostgreSQL [63], Oracle Databese [64], IBM DB2 [65] などがある．

一方，KVS では，保存データとそれを取得するためのキーを用いデータを管理する．KVS では一般的に RDBMS のような複雑なデータ操作は提供していないが，保存データのパーティショニングに適しておりサーバの追加による容量の拡張や I/O スループットの向上を RDBMS よりも容易に実現することができる．したがって，複雑なデータ操作が不要でかつ大規模なデータセットを扱うことの多いウェブサービスに適しており，近年多くの KVS が利用されている．このような KVS には，たとえば，Bigtable [66,67], HBase [68], Hypertable [69], Dynamo [70,71], Cassandra [72], MongoDB [73], Oracle NoSQL Database [74], Azure Table Storage [75], Redis [76], LevelDB [77], Kyoto Cabinet [78] などがある．

1.3 本研究の課題

本節では，Flash Crowds 発生時にウェブサービスを安定して配信するために，対策手法が解決しなければならない課題について整理する．まず，Jung らおよび Freedman による過去の実例報告 [17,79] を用いて Flash Crowds の性質を定量的に明らかにし，対策手法の設計についての指針を与える．続いて，この解析結果を基に，Flash Crowds 対策手法が備えるべき性質について説明する．

1.3.1 Flash Crowds の性質

本節では，Jung らおよび Freedman による過去の実例報告 [17,79] を用いて Flash Crowds の性質を明らかにし，対策手法の設計についての指針を与える．1) 規模，2) 増加速度，3) 継続時間，4) 対象の局所性の 4 つの観点から Flash Crowds を定量的に解析する．Flash Crowds の予測不可能性は，予測に基づくアプローチによる対策を困難にさせる．しかしその一方で Flash Crowds は，対策手法の設計に有利に働く性質も持っている．本論文における提案手法は，この解析に基づくものである．

規模

Jung らは過去に実際に発生した 2 件の Flash Crowds の実例について，ウェブサービスのアクセスログを用いた解析結果を報告している．一方は人気テレビ番組に連動したウェブサイトで，もう一方は 1999 年に行われたチリ大統領選挙の速報サイトである．最初の例では，4,100 s の間に，毎秒数リクエストであったリクエスト頻度（一秒間あたりのリクエスト数，rps）が 6,719 rps まで増加した．第二の例では，40 s の間に，0 rps から 300 rps に増加した．また，Freedman は CoralCDN [80, 81] を用いた計測より，Flash Crowds 発生時のリクエスト頻度の増加を解析した．CoralCDN とは，ウェブサーバの負荷分散に利用される Peer-to-Peer 型のコンテンツ配信網である．CoralCDN の説明は 第 2.1.1 節に詳しい．この解析により Freedman は，Slashdot effects の実例として `slashdot.org` からの遷移によるリクエストが約 60 分間で 0 rps から約 250 rps に増加した事例を示した．

これらの事実は，Flash Crowds によるリクエスト頻度の増加は事例により異なることを示す．したがって，予測に基づくアプローチを用いる場合，サービス提供者は Flash Crowds 発生時に生じうる負荷の最大値を何らかの手段で見積もらなければならない．また，予測に基づかないアプローチを用いる場合であっても，対応範囲を広げるために利用可能な資源量の上限はより大きい必要がある．

増加率

Flash Crowds によるリクエスト頻度の増加は，発生からピークに到達するまでにかかる時間も予測が難しい．第 1.3.1 項で述べたように，Jung らは過去 2 件の実例を解析し，ピークに到達するまでにそれぞれ 4,100 s，40 s かかったと報告している．Freedman は，CoralCDN で発生した Flash Crowds について，計測期間である 2009 年 8 月 9–18 日の 10 日間を 5 秒間のタイムスライスに分割し，各タイムスライスにおけるリクエスト数の変化を解析し，以下の結果を報告している．1) 数十倍に増加するタイムスライスが計測期間の 1 % 以上存在するドメインは計測対象全体の 1.7 % であった．2) 数百倍に増加するタイムスライスが存在したドメインは一例のみで，そのような変化をしたタイムスライスは計測期間の 0.006 % であった．3) 数千倍に増加するタイムスライスは存在しなかった．

これらの事実は，Flash Crowds は一瞬でピークに到達するのではなく，いくらかの時間を要することを示す．すなわち，Flash Crowds の発生自体を予測せず，発生を受けてから動作するというアプローチでも有効であることを示す．

継続時間

Flash Crowds の継続時間は永続的ではない．Jung らの示した 2 つの事例では，Flash Crowds の継続期間はそれぞれ 100 分間および 282 分間であった．Freedman は CoralCDN を利用中の 2 つのウェブサイトで観測された Flash Crowds の様子を示した．第一の例では，2005 年 5 月に Slashdot に記事が掲載されたことで Flash Crowds が発生し，約 3 時間の後突然沈静化した．突然の沈静化について Freedman は，Slashdot のトップページから記事が消えたのではないかと推察している．第二の例では，2009 年 8 月にニュースサイト `reddit.com` [82] の関連サイトで発生した複数回の小規模な Flash Crowds について，ピークの継続が数分間であったことを示した．

これらの事実は，Flash Crowds は永続的ではなく，Flash Crowds 対策のために長期間に渡って大量の資源を稼働し続ける必要はないことを示す．

対象の局所性

Flash Crowds 発生時の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 らの示した 2 つの事例では，90 % 以上のリクエストがサイト内における人気上位 10 % 以内のコンテンツに集中していた．また，Freedman は，CoralCDN が処理したリクエストの 49.1 % が人気上位 0.01 % のコンテンツ，92.2 % が人気上位 10 % に集中していたことを示し，またそれらの合計データサイズはそれぞれ 14 MB，28.7 GB であったと報告した．

これらの事実は，一部の限られた種類のコンテンツがリクエストの大部分を占めておりキャッシュが Flash Crowds によるサーバの負荷を効果的に減少させ得ることを示す．

1.3.2 Flash Crowds 対策手法の要件

一般に，ウェブサービスへのアクセス集中には，サービスコアが過負荷となることを抑制することが必要である．サービスコアに障害が発生するとウェブサービスを継続できなくなるため，サービスコアを構成する各コンポーネントが過負荷となることを防ぐ必要がある．サービスコアの過負荷を防ぐ方法は，1) 負荷軽減型および 2) 負荷分散型の 2 つに大別できる．第 1.4 節にて述べたように，負荷軽減はサービスコアに到達するリクエスト数を減少させることで，負荷分散はサー

ビスコアに到達したリクエストの処理を同じ機能を持つ複数のコンポーネントに分散させることである．ウェブサービスにおける Flash Crowds 対策としてこれらの方法を用いる場合，どちらか一方ではなく両方を適用する必要がある．なぜならば，第 1.3.1 項で述べたように，それぞれ配信に適したコンテンツが異なるためである．

Flash Crowds 発生時にウェブサービスを安定して配信するためには，Flash Crowds の特性を考慮して，負荷軽減および負荷分散する必要がある．Flash Crowds は第 1.3.1 節にて述べたように緩慢な需要変動とは異なる特性を持つ．本論文では，Flash Crowds に対応するために必要な 4 つの性質を a) 拡張性，b) 敏捷性，c) 伸縮性，d) 一貫性と定義し，以下，各性質の定義について詳細に述べる．

拡張性は，“資源の増加量に対し効率的に処理能力が増加する性質” と定義する．Flash Crowds 発生時は平常時に比べ負荷が数百～数千倍となるため，必要な資源量も増大する．ここで，追加する資源の量に対して得られる処理能力の向上が小さい場合，莫大な量の資源を必要としてしまうため非効率的である．たとえば，拡張性の高い手法では資源量を 10 倍にすることで 10 倍の処理能力が得られるところ，拡張性の低い手法では資源量を 20 倍にしないと同等の処理能力を得られない．Flash Crowds に対応するためにはより高い処理能力が必要となるため，拡張性の高さは効率的な資源利用のために重要である．

敏捷性は，“負荷の増加に対して遅れることなく処理能力を向上させる性質” と定義する．Flash Crowds は発生からピークに到達するまでに，短い場合で数分程度である．したがって，Flash Crowds が発生しアクセス数が増加し始めると同時に処理能力も増大させる必要がある．処理能力の増加が負荷の増加に遅れるとサーバが過負荷となりサービス障害に繋がるため，高い敏捷性が必要である．

伸縮性は，“負荷に対して適切に処理能力を増減させる性質” と定義する．Flash Crowds 発生時の負荷は平常時の数百～数千倍であるため，それに対応できる量の資源が利用可能である必要がある．なぜなら，必要量の資源が確保できない場合，サーバは過負荷となりサービス障害に繋がるためである．また，必要量の資源を確保できる場合であっても Flash Crowds に対応するために使用される資源の量は平常時よりも多く，その運用コストも大きい．一方で，それらの資源を平常時においても稼働させることに有用性はない．したがって，Flash Crowds 発生時以外は余剰資源の稼働を避け，必要最低限の資源量でサービスを運用できることが望ましい．

一貫性は，“クライアントが取得するコンテンツの内容がオリジナルコンテンツ

の更新に遅れることなく更新される性質” と定義する．Flash Crowds による負荷を分散させるために，サーバの複製やキャッシュの配信を行っている場合，それらが配信する内容はオリジナルコンテンツと同等であることが望ましい．なぜなら，各クライアントによるリクエストはオリジンサーバに到達することなく処理されるため，クライアントはオリジナルコンテンツの更新内容を知ることができないためである．したがって，コンテンツの更新がなされた場合には，それ以降に取得されるコンテンツにおける矛盾を防ぐために，更新内容を複製サーバやキャッシュへ早急に反映する必要がある．一方で，複製サーバの情報やキャッシュの頻繁な更新はサーバの負荷増大の原因となるため，更新状況と併せて適切な設定を行う必要がある．

以上のように，Flash Crowds 発生時にもウェブサービスを安定して提供するためには，1) サービスコアへのアクセス数を減少させ，2) サービスコアの処理を分散させるとともに，それぞれ a) 拡張性，b) 敏捷性，c) 伸縮性，d) 一貫性を備える必要がある．

1.4 本研究の目的

本研究では，Flash Crowds のような急激な需要変動が発生した場合においてもウェブサービスを安定的に配信する手法の提案を目的とする．Flash Crowds 発生時にウェブサービスを安定的に配信するということは，Flash Crowds 発生時でもサービスコアを構成する各コンポーネントに対する負荷が許容量を超えないようにすることである．各コンポーネントに対する負荷の増加を抑えるため，本研究ではサービスコアに対する負荷軽減および負荷分散の両方を実現する．

負荷軽減とは，サービスコアに到達するリクエスト数を減少させることで各コンポーネントに対する負荷を抑えることである．適切に負荷軽減できている場合，Flash Crowds によって需要が増加した場合もサービスコアにおける処理量の増加が小さいため，各コンポーネントが過負荷になることを抑制できる．このアプローチでは，サービスコアで本来行われる処理を行わずキャッシュによってリクエストを処理するため，ひとつのリクエストの処理に必要な計算資源を小さく抑えることができる．しかし，本来サービスコアで行う処理を行わないため，静的コンテンツに対するリクエストおよびべき等性がありサービスの状態を変更しないリクエストにのみ対応可能である．

一方負荷分散とは，サービスコアに到達したリクエストの処理を同じ機能を持つ複数のコンポーネントに分散させることである．適切に負荷分散できている場合，Flash Crowds によって需要が増加した場合も各コンポーネントの負荷の増加は小さいため，過負荷になることを抑制できる．このアプローチでは，サービスコアが通常提供するサービスと同等のものを提供できるため，対象のウェブサービスで扱うコンテンツはすべて取り扱うことが可能である．しかし，計算資源を大量に用意する必要があり，また，同じ機能を持つコンポーネント間における同期や排他制御などのオーバヘッドが生じる．

本研究では，負荷軽減と負荷分散の両方を実現することにより，Flash Crowds 発生時においても安定的にウェブサービスを配信する手法を示す．なお，負荷軽減と負荷分散を実現するために本研究では，第 1.2 節で述べたウェブサービスの各コンポーネントのうち，次の 2 つのコンポーネントを対象とする．第一に，クライアント側コンポーネントを対象とする．クライアント側コンポーネントは Flash Crowds によるサービスコアに対する負荷の発生源であり，ここで負荷軽減の仕組みを取り入れることはサービスコアに対する負荷を確実に軽減するためである．第二に，サーバ側コンポーネントのストレージ層を対象とする．サービスコアを構成する 3 層のうち，ウェブ層およびアプリケーション層と異なりストレージ層のみ永続的な状態を持ち，また，上位 2 層はストレージ層に依存しているためである．永続的な状態を持たないウェブ層やアプリケーション層については，クラウド方式 [83–88] のように仮想マシン [89] などの技術を用いることによる，サーバの追加や削除に適した手法を適用する．

1.5 本研究の提案

本研究では，Flash Crowds のような突発的な需要変動が生じた場合であっても，安定的にウェブサービスを提供するための手法を提案する．図 1.4 に提案手法の位置付けを示す．図 1.4 において，赤色で示す部分がクライアント間連携による負荷軽減手法，青色で示す部分がデータセンタ間連携におけるストレージの負荷分散手法を指す．本手法は，負荷軽減および負荷分散の双方に対応することで，Flash Crowds によるサービスコアの過負荷を抑制する．

負荷軽減は，キャッシュを共有するための Peer-to-Peer (P2P) ネットワークをクライアント間で構築することにより実現する．この P2P ネットワーク上では，各ク

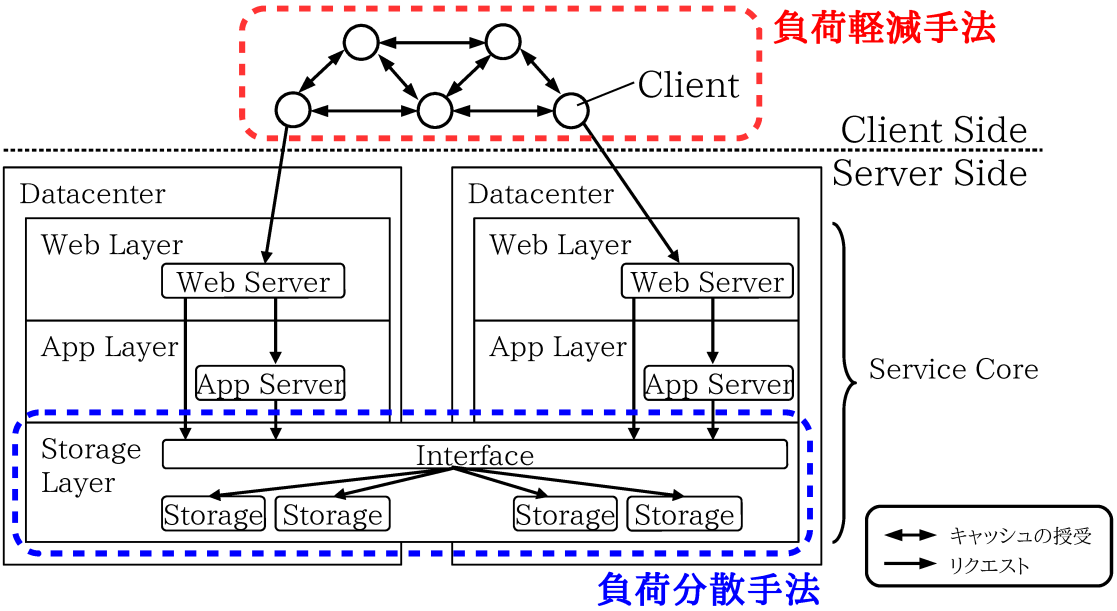


図 1.4: 提案手法の位置付け

ライアントが取得したコンテンツをキャッシュとして共有する．目的のコンテンツのキャッシュをこの P2P ネットワーク上から取得できたクライアントは，サービスコアへ直接リクエストすることなく目的のコンテンツを取得できるため，サービスコアに対して発行されるリクエスト数は減少する．Flash Crowds 発生時のように多くのリクエストが狭い範囲のコンテンツに対して集中している状況においては，多くのリクエストをキャッシュで処理することが可能となるため，サービスコアへのアクセス数を効果的に軽減させることができる．また，提案方式では，クライアントが取得したあらゆるコンテンツのキャッシュを共有するため，Flash Crowds 発生時には該当のキャッシュが既に利用可能な状態となっており，Flash Crowds の対象となるコンテンツの予測を必要としない．

負荷分散は，複数のデータセンタを跨いで利用可能な広域分散型キーバリューストア (KVS) を提供することで実現する．この KVS は，複数のデータセンタの資源を用いることで，単一のデータセンタを用いる場合に比べ，利用できる資源の量やその地理的な位置の自由度が増す．ウェブ層やアプリケーション層はストレージ層に依存するため，ストレージ層の自由度が増すことにより，サービスコア全体の負荷分散をより効果的にする．サービスコアのうちストレージ層に着目する理由は 2 つある．第一に，ウェブ層やアプリケーション層はストレージ層に依存しており，ストレージ層の重要性が高いためである．ストレージ層における

負荷分散が不十分であると，ストレージ層がボトルネックとなりウェブサービスの品質を低下させてしまう．第二に，ストレージ層はウェブ層やアプリケーション層と異なり，負荷分散のためのコンポーネント追加に際して制約が強いためである．ストレージ層は他の 2 層と異なり永続的なデータを持つため，複製を配置する際にはそれらの一部または全部も含める必要がある．一般にウェブサーバやアプリケーションサーバのデータサイズよりも，ストレージに保管されているデータサイズの方が大きいため，ストレージ層における負荷分散は必要な時間や消費する計算資源も大きい．以上の理由により，本研究ではサービスコアの負荷分散にあたり，ストレージ層に着目する． 複数のデータセンタのサーバ資源を用いることで，サーバの配置の自由度を向上させることが可能となる．既存手法においてサーバの配置はデータセンタ内で閉じていたが，本手法を用いることで任意のデータセンタにウェブサーバやアプリケーションサーバを配置することが可能となる．

1.6 本研究の貢献

本研究の貢献は，Flash Crowds 発生時においても安定的にウェブサービスを配信するための手法を負荷軽減と負荷分散の両面から提供することである．これにより，ウェブサービスの提供者は，運用するサービスの需要に合わせて適切な量の資源を利用することができる．

表 1.1 に本研究の貢献をまとめる．提案手法は，負荷軽減および負荷分散の双方について，第 1.3.2 項で述べた 4 つの性質を満たす．一方，既存の Flash Crowds 対策手法は，1) プロキシ方式, 2) キャッシュ共有方式, 3) クラウド方式, 4) 完全複製方式 の 4 つに大別できるが，4 つの性質をすべては満たすことができない．プロキシ方式は，キャッシュの配信によりサービスコアの負荷を軽減するが，予め用意されたプロキシサーバを運用するため急激な需要変動に対応できない．キャッシュ共有方式は，クライアントの資源を用いるため，需要変動に応じて資源量が自然に増減するが，キャッシュを広く分配するためコンテンツの更新反映に時間を要する．クラウド方式は，需要変動を検知してサーバを複製するが，データセンタの規模の制約を受ける．完全複製方式は，複数のデータセンタの資源を用いることでより多くの複製を生成可能であるが，個別のデータセンタの規模の制約を受け，また，データセンタの追加は全データの複製を要するため多大な時間を要す

表 1.1: 本研究の貢献

		拡張性	敏捷性	伸縮性	一貫性
負荷軽減	プロキシ方式	✓	×	×	✓
	キャッシュ共有方式	✓	✓	limited	×
	提案手法: MashCache	✓	✓	✓	✓
負荷分散	クラウド方式	✓	conditional	limited	✓
	完全複製方式	✓	×	×	conditional
	提案手法: Pangaea	✓	✓	✓	✓

る．既存手法についての説明は，第 2 章に詳しい．

1.7 本論文の構成

本論文は全 5 章からなる．第 1 章では本研究の背景，動機および目的について述べ，本研究の学術的貢献について説明した．第 2 章では本研究の関連研究をまとめる．既存研究を負荷軽減と負荷分散の 2 つに大別し，それぞれについてプロキシ方式と分割配信方式およびクラウド方式とサービス複製方式の 2 つずつに整理し，本研究との差分を明らかにする．第 3 章, 第 4 章では本研究の提案手法である負荷軽減手法および負荷分散手法について説明をする．第 3 章では，クライアント間連携による負荷軽減手法である MashCache について述べる．MashCache の設計およびその実現方法として実装について述べた後，Flash Crowds を模したシミュレーションにより MashCache が Flash Crowds 対策として有効であることを定量的に評価する．また，MashCache のオーバヘッドについての議論も行う．第 4 章では，データセンタ間を跨いで利用可能なストレージの負荷分散手法である Pangaea について述べる．Pangaea の要件，設計および実装について述べた後，シミュレーションおよび実際のインターネット環境を用いた実験により，データセンタ間を跨ぐことによる性能低下の改善について評価を行う．また，Pangaea の要素技術について関連する手法との定性的な比較を行う．最後に第 5 章で本論文をまとめる．

第2章　関連研究

本章では，既存の Flash Crowds 対策について述べる．既存の Flash Crowds 対策手法を 4 つの方式に分類し，各方式について第 1.3 節で挙げた 4 つの性質について定性的に評価をする．

2.1　負荷軽減手法

負荷軽減手法は，サービスコアとは独立してコンテンツのキャッシュを配信し，サービスコアの負荷を抑制する方法である．本手法では，各クライアントからのリクエストのすべてまたは一部はサービスコアに到達することなく処理が完了する．サービスコアとは独立してコンテンツのキャッシュを配信するため，リクエストごとにサービスコアでの処理が必要でない静的コンテンツが主な対象となる．本項では，負荷軽減手法をプロキシ方式およびキャッシュ共有方式の 2 つに大別し，それぞれの特徴について整理する．

2.1.1　プロキシ方式

プロキシ方式は，オリジナルコンテンツを保持するサービスコアの代わりにプロキシサーバからコンテンツのキャッシュを配信する方法である．図 2.1 にプロキシ方式の構成例を示す．各クライアントは，サービスコアではなくプロキシサーバからキャッシュを取得することで目的のコンテンツと同じデータを取得することができる．本方式では，プロキシサーバが処理したリクエストはサービスコアへ到達しないため，Flash Crowds によって負荷が増加している場合であってもサービスコアが過負荷になることを防ぐことができる．一度プロキシサーバに配置されたキャッシュは破棄されるまで後続のクライアントに提供されるため，Flash Crowds のように一部のコンテンツに対してリクエストが集中する場合には少量のキャッシュで効果的にサービスコアの負荷を軽減することができる．

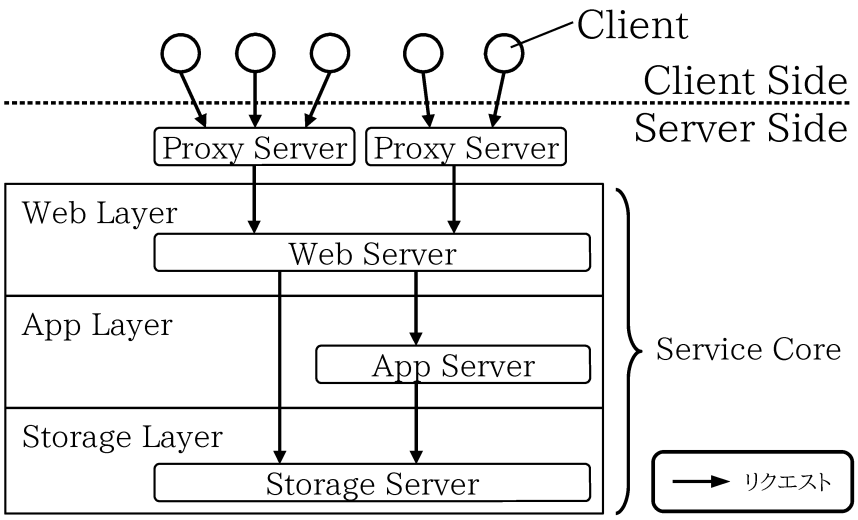


図 2.1: プロキシ方式の構成例

また，本方式の利点は 2 つある．第一に，コンテンツプロバイダにとってキャッシュの管理が容易であることである．コンテンツプロバイダがキャッシュとしてプロキシサーバに配置するコンテンツを選択することが可能であり，また，必要に応じてキャッシュの破棄や更新を行うことができる．また，プロキシサーバの台数を追加することで，より多くのリクエストを処理することが可能である．第二に，クライアントに対して透過的である．本方式は，クライアント側コンポーネントへの改変は含まないため，各クライアントに特別なソフトウェアの導入などを前提としない．

一方，Flash Crowds 対策として用いる場合，本方式には欠点も 2 つある．第一に，Flash Crowds はその規模や発生時期の予測が困難であるため，適切なプロキシサーバの台数の見積もりも困難である．プロキシサーバの台数が不足する場合，Flash Crowds による負荷を十分軽減できず，サービス障害に繋がる．実際，本方式を用いた商用サービスの Akamai [90] は，リクエスト殺到による過負荷でサービス障害が発生したことがある [13]．一方プロキシサーバの台数が過剰である場合，過剰にプロキシサーバを稼働させるためのエネルギーや運用コストを浪費する．これは Flash Crowds が発生していない間も同様である．

Squid [91] はサービスコアに代わってコンテンツのキャッシュをクライアントに提供するプロキシサーバソフトウェアである．プロキシサーバとしての基本的な機能を提供する．

Backslash [92] は複数のプロキシサーバを分散ハッシュ表 (DHT) を用いて相互に接続しオーバレイネットワークを構築する．クライアントからのリクエストを受け付けたプロキシサーバは，オーバレイネットワーク上でキャッシュの探索を行う．本手法では，プロキシサーバを個別に用いる場合と比べ，多くの資源を柔軟に利用することが可能である．

Akamai [90], Orchestrate Delivery Network [93], CloudFront [94] は商用のコンテンツ配信網である．これらのコンテンツ配信網では，世界中に配置されたエッジサーバと呼ばれるプロキシサーバを通じてキャッシュを配信する．サービスコアに対するクライアントのリクエストは Domain Name System (DNS) リダイレクトによりエッジサーバに転送され，クライアントはエッジサーバよりキャッシュを取得する．DNS リダイレクトとは，DNS サーバのレコードを変更し，Fully Qualified Domain Name (FQDN) に紐づく IP アドレスを別の IP アドレスと置き換えることにより，ネットワーク接続を任意のサーバへ透過的にリダイレクトする方法である．エッジサーバはサービスコアと専用線などの高速なネットワークで接続されており，コンテンツデータを低遅延かつ高スループットでエッジサーバに配置することができる．また，エッジサーバは一般的に，インターネットサービスプロバイダ (ISP) 単位などで設置されており，各エッジサーバと末端のクライアントとのネットワーク上における距離は近い．そのため，エッジサーバを経由してクライアントにコンテンツの配信を行うことは，サービスコアからクライアントへ直接コンテンツを配信する場合に比べ，より小さい遅延かつより高いスループットを得ることが可能である．しかし，エッジサーバの設置やサービスコアとエッジサーバ間のネットワーク接続は静的に構築するため，Flash Crowds 対策として用いるには予測が前提となる．したがって，想定を超える規模の Flash Crowds が発生した場合や，想定と異なる地域 (ISP) で Flash Crowds が発生した場合は，サービスコアの負荷を十分に軽減することができない．

CoralCDN [80,81] は分散管理型のコンテンツ配信網である．CoralCDN では，商用のコンテンツ配信網のように専用線などを用いず，DHT のひとつである Kademlia [95] を用いてエッジサーバ群を相互に接続し管理する．専用線を用いる場合に比べ通信遅延やスループットの点で不利であるが，需要に合わせてエッジノードを追加や削除することが容易である．通信における不利の対策として，CoralCDN ではサービスコアのみでなく別のエッジノードからもキャッシュを配置する．この際，Distributed Sloppy Hash Table (DSHT) [96] を用い Kademlia 上のノード群を通信遅延を基準にクラスタリングし，通信遅延の小さいエッジノード優先的に選択

表 2.1: プロキシ方式を採る手法の定性的比較

	拡張性	敏捷性	伸縮性	一貫性
Squid [91]	✓	×	×	✓
Backslash [92]	✓	×	×	×
Akamai [90]	✓	×	×	✓
Orchestrate Delivery Network [93]	✓	×	×	✓
CloudFront [94]	✓	×	×	✓
CoralCDN [80, 81]	✓	×	×	×

する．

プロキシ方式の特徴を表 2.1 にまとめる．いずれの手法も，プロキシサーバの追加に比例して処理できるリクエスト数が増加するため，拡張性は高い．一方，いずれの場合も静的に資源を準備しておく必要があるため，敏捷性は低い．伸縮性および一貫性については手法によって異なる．Squid, Akamai, Orchestrate Delivery Network, CloudFront では，資源の管理も静的に行うため伸縮性は低い．一方配置するプロキシサーバが明示的であるためオリジナルコンテンツの更新の反映は比較的容易であり一貫性は高い．Backslash, CoralCDN では，プロキシサーバを DHT に動的に追加することが可能であるため，他の手法よりは資源の追加や削除が容易で伸縮性を向上する余地がある．しかし，追加する資源は予め見積もる必要があるため，負荷に応じて資源を確保する仕組みと併用しなければ伸縮性を発揮することはできない．また，これらの手法ではキャッシュの配置場所が動的に移り伝播していき更新反映により時間がかかるため，一貫性は低い．

2.1.2 キャッシュ共有方式

キャッシュ共有方式は，コンテンツの配信にクライアントの資源を利用する方式である．図 2.2 にキャッシュ共有方式の構成例を示す．本方式では，各クライアントが取得したコンテンツの全部または一部をキャッシュとして他のクライアントと共有する．コンテンツの需要増加は利用可能な資源の増加を意味するため，潜在的に利用可能な資源の量という観点では本方式は Flash Crowds に適している．しかし，その資源を利用して多数のクライアントやキャッシュを効率的に管理する仕組みが必要であり，その方法によって特性が異なる．また，本方式は管理方法に

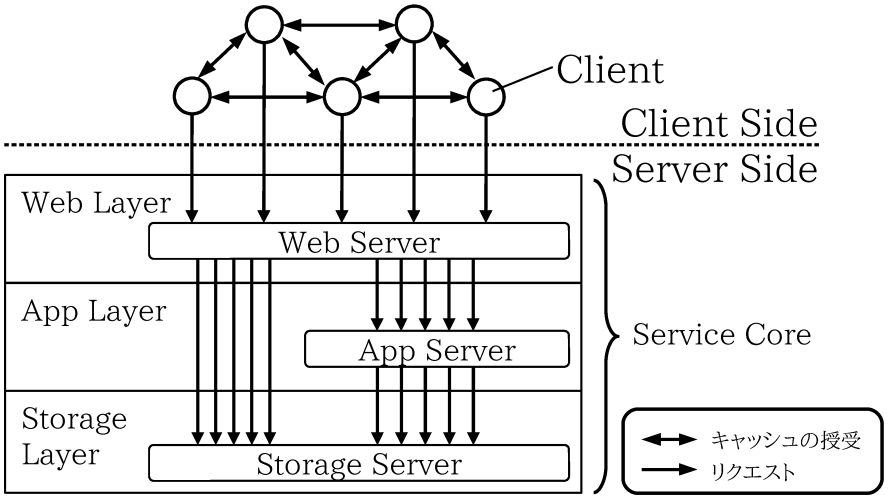


図 2.2: キャッシュ共有方式の構成例

よって 1) 集中管理型および 2) 分散管理型の 2 つに大別できる．

第一に，集中管理型は管理用サーバによりシステム全体を管理する方法である．この方法では，どのクライアントがどのコンテンツのキャッシュを保持しているかという情報などを集中管理する．各クライアントは，管理サーバより目的のキャッシュを保持するクライアントの接続情報を得る．

第二に，分散管理型は各クライアントが協調することで管理する方法である．この方法では，各クライアントが保持するキャッシュの情報を相互に授受することにより，目的のキャッシュを探索できるようにする．

BuddyWeb [97], BitTorrent [98], Avalanche [99], Flashback [100] は集中管理型の手法である．BuddyWeb, は，各クライアントが保持するローカルキャッシュを他のクライアントに提供する手法である．本手法では，各クライアントがどのコンテンツのキャッシュを提供できるかという情報を専用サーバで集中管理する．コンテンツの取得に際して各クライアントはサーバに問い合わせ，キャッシュを保持しているクライアントが存在する場合はそちらからキャッシュを取得する．キャッシュを提供するクライアントの負荷を考慮しておらず，利用可能な資源が増加しても分散しづらいため，拡張性が低い．

BitTorrent, Avalanche, Flashback は，各クライアントに対してコンテンツデータの断片を提供することで，サーバにおける 1 クライアントあたりのネットワーク帯域の消費量を軽減する方式である．本手法では，断片や交換相手となる他クライアントの接続情報はサーバより配信する．断片と接続情報を得たクライアントは相互

に接続し，目的のコンテンツを取得するために断片を互いに交換しあう．たとえば，あるコンテンツを 100 分割して配信する場合，1 クライアントのために消費するネットワーク帯域はおよそ 1/100 程度となる．ここで，通常であればサーバのネットワーク帯域を消費することで配信していた分は，各クライアントのネットワークに転嫁される．しかし，利用しているクライアント数が多いほど，断片を所有するクライアントも増加するため各クライアントの負荷は抑制される．すなわち，本手法では，人気のあるファイルほどダウンロード速度が向上するため，Flash Crowds のように特定のコンテンツに需要が集中する場合には適している．BitTorrent は特に大容量データの配信に広く用いられており，日本の大手ゲーム会社コナミがオンラインゲームソフトの更新データの配信に用いたとの報告がある [101]．Avalanche は，BitTorrent と類似しているが，断片の共有の際に Network Coding [102] を用いることで，どのノードがどの断片を持っているかを考慮せず効率的に収集することを可能にする．また，Flashback は一般的なデータサイズのウェブコンテンツを対象としており，断片を共有の仕組みをウェブページに含めることで自動的に共有の仕組みを利用可能である．

Squirrel [103] および Linga らの手法 [104] は分散管理型の手法である．Squirrel は，同じ Local Area Network (LAN) 内のクライアント間でローカルキャッシュを共有する手法である．本手法はウェブサービスの負荷を軽減するためではなく，LAN から目的のキャッシュを取得できるようにすることでインターネットアクセスを行うことなくコンテンツを取得可能にする．しかし，本手法によって得られる負荷軽減の効果は特定の LAN 内に属するクライアントの分だけである．また，オリジナルのコンテンツの更新への追従も考慮されていない．

Linga らの手法は，LAN でなくインターネット上のクライアント間でキャッシュを共有する手法である．本手法は，ネットワーク上における距離の近いクライアント同士をグループ化することで，キャッシュ探索時の通信遅延の短縮やキャッシュヒット率の向上を実現している．しかし，本手法も Squirrel と同様にコンテンツ取得に要する時間の短縮を目的としており，Flash Crowds のように特定のコンテンツに負荷が集中している場合にクライアントの負荷を分散することや，オリジナルのコンテンツの更新への追従も考慮されていない．

キャッシュ共有方式の特徴を表 2.2 にまとめる．BuddyWeb, Squirrel, Linga らの手法は各クライアントの負荷を考慮しておらず，需要増加に伴い潜在的に利用可能なクライアント資源が増加しても，それらを効率的に活用できないため拡張性および敏捷性は低い．一方 BitTorrent, Avalanche, Flashback は，キャッシュを断片化

表 2.2: キャッシュ共有方式を採る手法の定性的比較

	拡張性	敏捷性	伸縮性	一貫性
BuddyWeb [97]	×	×	limited	×
BitTorrent [98]	✓	✓	limited	×
Avalanche [99]	✓	✓	limited	×
Flashback [100]	✓	✓	limited	×
Squirrel [103]	×	×	✓	×
Linga らの手法 [104]	×	×	✓	×

するなどして各クライアントへの負荷を分散することにより，需要増加に伴って増加するクライアント資源を効率的に利用できるため，拡張性および敏捷性は高い．分散管理型の Squirrel, Linga らの手法は，使用する資源の変化は需要の変化と連動するため伸縮性は高い．一方，集中管理型の BuddyWeb, BitTorrent, Avalanche, Flashback は需要増加による管理サーバの負荷の増大が許容量を超えない範囲に限っては，クライアントの資源を活用できるため伸縮性は高い．一貫性はいずれの手法でも低い．なぜならば，キャッシュが多数のクライアントに出回ってしまい，更新の伝播が遅れるためである．

2.2 負荷分散手法

負荷分散手法は，サービスコアを構成する各ソフトウェアコンポーネントを複製することで，各コンポーネントの負荷を抑制する方法である．本手法では，各クライアントからのすべてのリクエストをサービスコアで処理する．負荷軽減手法と異なり，サービスコアでリクエストの処理を行うため，本手法を用いない場合と完全に同等の処理を行うことができる．そのため，静的コンテンツのみならず動的コンテンツも含めて対応可能である．本項では，負荷分散手法をクラウド方式および完全複製方式の 2 つに大別し，それぞれの特徴について整理する．

2.2.1 クラウド方式

クラウド方式は，各サーバの負荷状況を監視し，負荷がしきい値を超えた場合にサーバの数や割り当て資源量を増減させる方式である．図 2.3 にクラウド方式の

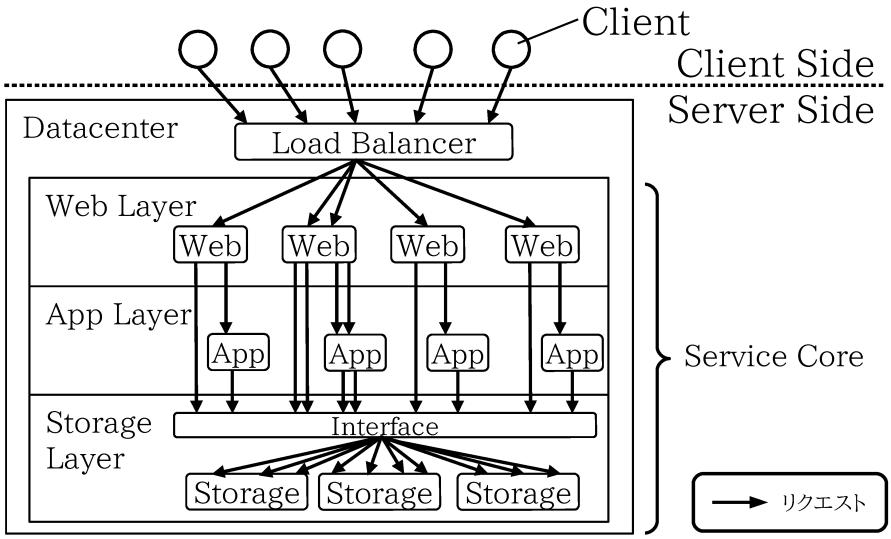


図 2.3: クラウド方式の構成例

構成例を示す．本方式は，テナント（サービス事業者）に対して提供する資源のレイヤで大別される．テナントに対し，仮想マシンを提供し Amazon Elastic Compute Cloud (Amazon EC2) [83], CloudStack [84], Eucalyptus [85] 等の Infrastructure as a Service (IaaS) では，主に仮想マシン [89] が提供され，テナントは割り当てられた仮想マシンを自由に運用することができる．Google App Engine [86], Microsoft Azure [87], Heroku [88] 等の Platform as a Service (PaaS) では，定められたソフトウェアフレームワークのみが動作する環境が提供され利用の範囲が制限されるが，環境構築や各種ソフトウェアのアップデートが自動で適用される等管理コストをクラウド事業者側が負っている．

本方式において一般的に各テナントは，監視する資源およびその状況に応じた処理を設定することができる．例えば，”過去 5 分間の平均 CPU 使用率が 80 % を超えていたらインスタンスを 2 つ増加させる” といった設定である．Flash Crowds が発生した場合に顕著な変化が想定される資源について適切な監視間隔やしきい値の設定がなされていれば，Flash Crowds に伴って負荷が増大した場合にも適切に資源が増強され，サービス障害を引き起こすことなく Flash Crowds に対応することが可能である．適切なしきい値の設定がなされている場合，本方式は高い敏捷性を発揮する．しかし，適切なしきい値は，稼働するアプリケーションの特徴やワークロードの変化などに大きく影響されるため，最適な設定を保持することは困難である．

表 2.3: クラウド方式を採る手法の定性的比較

	拡張性	敏捷性	伸縮性	一貫性
Amazon EC2 [83]	✓	conditional	×	✓
CloudStack [84]	✓	conditional	×	✓
Eucalyptus [85]	✓	conditional	×	✓
Google App Engine [86]	✓	conditional	×	✓
Microsoft Azure [87]	✓	conditional	×	✓
Heroku [88]	✓	conditional	×	✓

クラウド方式の特徴を表 2.3 にまとめる．いずれの手法も，インスタンスの追加にほぼ比例して処理できるリクエスト数が増加するため，拡張性は高い．敏捷性については，適切な値を設定できた場合には高いが，適切な値の設定にはどの資源のどの値を監視するかなど，サービスの特性を熟知している必要がある．また，本方式は，高速なネットワークで接続されたストレージを用いるため一貫性は高い．一方，いずれの場合も利用可能な資源単一のデータセンタに限定されており，伸縮性は低い．近年，単一のデータセンタの資源ではウェブサービスの運用に十分でないとの報告があり，複数のデータセンタの資源を用いる手法が必要とされている [105].

2.2.2 完全複製方式

完全複製方式は，対象とするウェブサービスのレプリカを各データセンタに配置し，ストレージの内容を相互に同期する方式である．図 2.4 に完全複製方式の構成例を示す．一般的なウェブサービスの場合，ウェブサーバ，アプリケーションサーバ，データストレージ等の全てのサービスコンポーネントが，各データセンタで稼働する．各データセンタには全てのサービスコンポーネントが存在するため，あるデータセンタが利用不能になった場合でも継続して運用可能である (Disaster Recovery)．しかし，各データセンタがそれぞれ完全に独立稼働していることは保存データの一貫性に不整合が生じ得るが一般的に厳密な同期を取るシステムは少ない．なぜならば，別のデータセンタとの通信は応答時間やデータ転送に要する時間時間がかかるためである．Walter [106], COPS [107], Spanner [108, 109], Gemini [110] はそれぞれ一定の同期レベルを保持する．

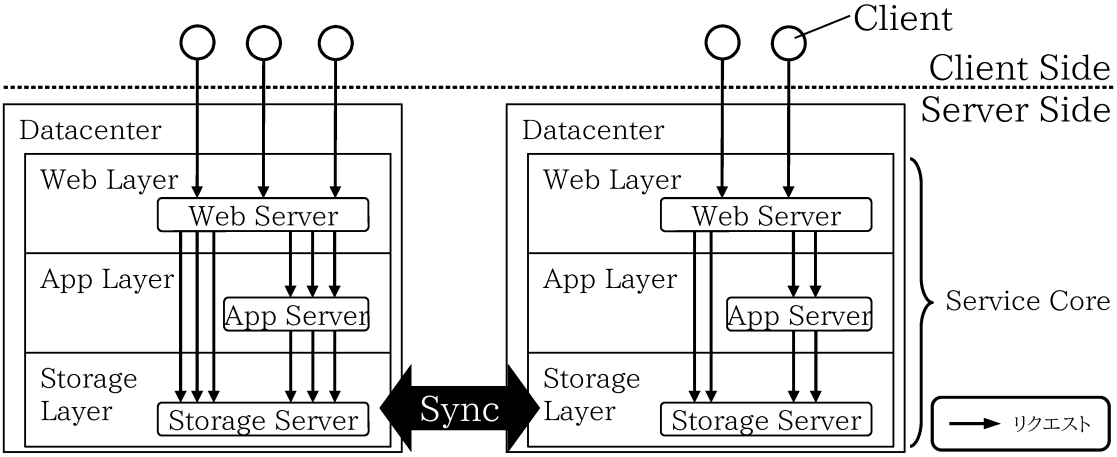


図 2.4: 完全複製方式の構成例

表 2.4: 完全複製方式を採る手法の定性的比較

	拡張性	敏捷性	伸縮性	一貫性
Walter [106]	✓	×	×	conditional
COPS [107]	✓	×	×	conditional
Spanner [108, 109]	✓	×	×	conditional
Gemini [110]	✓	×	×	conditional

本方式は，拠点が増加すると利用可能な資源量も増大するため拡張性が高い．一方で，需要が偏在している場合でも，各拠点で一定以上の資源を稼働させる必要があるため，伸縮性は低い．新たにデータセンタを追加する場合は複製の生成に時間が必要であるため，敏捷性は状況に依存する．また，一貫性は同期レベルに依存するが，同期レベルと応答遅延はトレードオフの関係にあるため，ワークロードの特性と要求性能を基に適切な手法を選択する必要がある．

完全複製方式の特徴を表 2.4 にまとめる．いずれの手法も，データセンタの追加に比例して処理できるリクエスト数が増加するため，拡張性は高い．一方，データセンタの追加にはストレージ内の全データの転送を要するため敏捷性は低い．複数のデータセンタに属する資源を利用可能ではあるが，各データセンタにおいて使用する資源の規模は各複製を最低限運用可能な水準で稼働させる必要があるため，伸縮性は低い．また，一貫性は，データ更新時におけるの同期レベルに依存する．例えば，Eventual Consistency では，リモートデータセンタとの同期を後回

しにするため，応答性は高いが一貫性が低い．一方，Strong Consistency では，リモートデータセンタとの同期完了を待つため，応答性は低いが一貫性は高い．

2.3 まとめ

本章では, 本研究の関連研究を 1) プロキシ方式, 2) キャッシュ共有方式, 3) クラウド方式, 4) 完全複製方式 の 4 種類に分類し，第 1.3 項で挙げた Flash Crowds 対策の 4 つの要件 a) 拡張性, b) 敏捷性, c) 伸縮性, d) 一貫性を満たすことは難しいことを確認した．本研究ではこれら 4 つの要件すべてを満たす負荷分散手法および負荷軽減手法を提案する．第 3 章および第 4 章 で提案手法の詳細について述べる．

第3章　クライアント間連携による負荷軽減手法

本章では，クライアント間連携によってサービスコアの負荷を軽減する手法 MashCache について述べる．本手法は，クライアント間で構築するキャッシュ共有ネットワークを用いることで，Flash Crowds 発生時におけるリクエストの多くをサービスコアに到達させることなく処理する．Flash Crowds の負荷に耐えるため，本手法は分散ハッシュ表 (DHT) を用いた完全分散型の構造を採用する．また，Flash Crowds の特性に基づき設計した 1) Aggressive Caching, 2) Query Origin Key, 3) Cache Meta Data, 4) Two-phase Delta Consistency の 4 つの要素技術により Flash Crowds への対応を実現する．本章では，まず，各要素技術の設計について説明する．次に，実装について説明し，本手法の効果をシミュレーション環境を用いた実験により評価する．その後，本手法のオーバヘッド，セキュリティ，キャッシュヒット率，一貫性について議論を行い，最後に本章をまとめる．

3.1　設計

本節では，クライアント側で動作する負荷軽減手法である MashCache の設計について述べる．まず，MashCache の概要を説明し，Flash Crowds 対策としての位置付けを明らかにする．続いて，MashCache を構成する 1) Aggressive Caching, 2) Query Origin Key, 3) Cache Meta Data, 4) Two-phase Delta Consistency の 4 つの要素技術について説明し，MashCache の設計を示す．

3.1.1　概要

MashCache は第 1.3.1 項における Flash Crowds の解析に基づく，ウェブサービスの負荷軽減手法である．本手法ではウェブサービスが配信するコンテンツをクライアント間で共有することで負荷軽減を実現する．MashCache を用いることで，

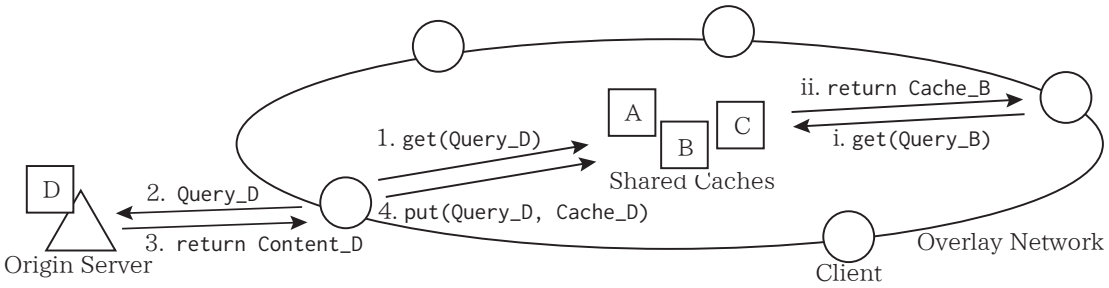


図 3.1: MashCache におけるコンテンツ取得の流れ

Flash Crowds 発生時のリクエストの多くをサービスコアに到達させることなく処理することが可能となる．

MashCache において，各クライアントは取得したコンテンツをキャッシュとして共有する．図 3.1 に MashCache によるコンテンツ取得の流れを示す．各クライアントは，通常時より，キャッシュ共有のためのネットワークに参加する．コンテンツを取得する際，クライアントはまずこの共有ネットワークからのキャッシュ取得を試みるが，キャッシュの有無により動作が異なる．目的とするコンテンツのキャッシュが存在する場合，図中 i, ii のように，クライアントはウェブサービスにリクエストを発行することなく目的のコンテンツを取得することができる．一方もし目的のキャッシュが存在しなかった場合は，図中 1–3 のように，オリジナルコンテンツを提供するウェブサービスにリクエストを発行し，目的のコンテンツを取得する．オリジナルコンテンツを取得したクライアントは，図中 4 のように，それを共有ネットワークに配置する．この 1–4 の処理により，以降同じコンテンツの取得を試みるクライアントは共有ネットワークからキャッシュを取得することができる．

また，本手法は完全にクライアント側で動作するため，サービスコアを始めとするサーバ側のコンポーネントに一切手を加える必要がない．各クライアントにおける MashCache の位置付けを図 1.4 に示す．MashCache は，キャッシュ共有ネットワークを完全分散型の P2P ネットワークとして構築する．完全分散型の構造を採用することにより，クライアントやキャッシュを管理するためのインデックスサーバなどの設置を避けることができる．また，このような特別なサーバの設置を避けることで，負荷集中や単一障害点が生じることを防ぐ．この共有ネットワークにより，MashCache は本ネットワークに参加しているクライアントの資源を集約し，単一のキャッシュ共有用ストレージとして機能させることができる．また，この方

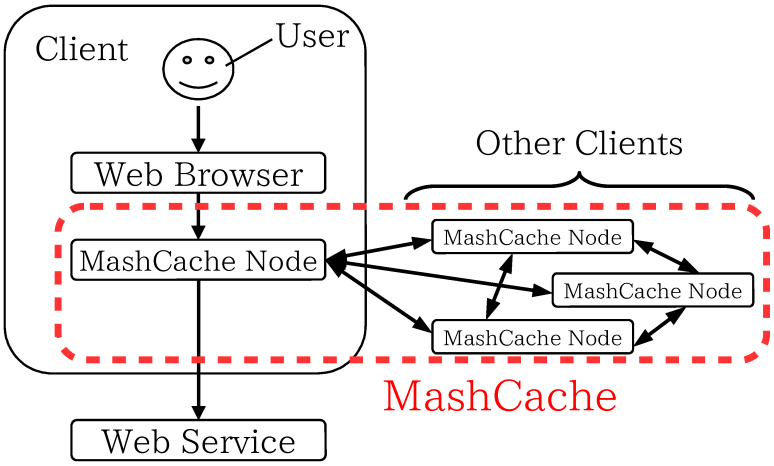


図 3.2: クライアントにおける MashCache の位置付け

法では Flash Crowds の発生時期や規模の予測を必要としない．なぜならば，Flash Crowds 発生による，あるコンテンツにアクセスするクライアント数の増加は，本ネットワークで利用可能な資源の増加を意味するためである．本研究で実装した MashCache のプロトタイプでは，完全分散型の P2P ネットワークを構築するために 分散ハッシュ表 (DHT) を採用する．

MashCache の共有ネットワークを構成するノードは各クライアントのローカルで動作する．図 3.2 に，クライアントにおける MashCache の位置付けを示す．MashCache ノードはローカルプロキシとして動作し，ウェブブラウザなどのウェブサービスを利用するアプリケーションからのリクエストを受け付ける．MashCache ノード同士は，DHT によって相互に接続されており，ウェブブラウザよりリクエストを受けた際は，リクエストからキャッシュを取得するためのキーを生成し，担当のノードを探索する．キーの生成方法およびキャッシュの取得方法については第 3.1.3 項および第 3.1.4 項に詳しい．生成したキーを用いてキャッシュを取得できた場合は，ウェブブラウザに対してキャッシュを提供しリクエスト処理を完了する．一方，キャッシュが存在しない場合は，直接ウェブサービスからオリジナルコンテンツを取得し，ウェブブラウザに提供する．この際取得したコンテンツはすべて，キャッシュとして共有ネットワークに配置する．また，共有キャッシュは，負荷軽減と保存データサイズの抑制を両立するために，一定の基準で破棄する．このキャッシュの生成および破棄のポリシーについては第 3.1.2 項および第 3.1.5 項に詳しい．

3.1.2 Aggressive Caching

MashCache ではキャッシュポリシーとして Aggressive Caching を採用する．Aggressive Caching の採用により MashCache は，Flash Crowds の予測をすることなく，Flash Crowds の対象となるコンテンツのキャッシュを配信可能とする．Aggressive Caching は，各クライアントが取得したあらゆるコンテンツをキャッシュとして保存し，短期間で破棄するポリシーである．すなわち，Aggressive Caching において各クライアントは，サービスコアから取得したあらゆるコンテンツをキャッシュ共有ネットワークに配置する．これにより，どのコンテンツに対して将来 Flash Crowds が発生するかいう予測が不要となる．

Aggressive Caching のようにクライアントが取得した後にすべてのコンテンツをキャッシュとして共有する場合，通常は 1) 人気が集中する前にキャッシュが準備されていない，2) キャッシュの量が膨大になり管理できないという 2 つの問題が生じ得るが，Flash Crowds に着目する場合，その特性によりこれらの問題は回避可能である．第一に，Flash Crowds は発生から一瞬でピークに到達するわけではなく，それまでに数十秒以上の時間を要する．すなわち，最初にそのコンテンツを取得したクライアントがキャッシュとして配置するために要する時間が存在する．Flash Crowds によりクライアント数が増加した場合，その対象となるコンテンツはリクエストの増加がピークを迎える前にキャッシュとして利用可能な状態となる．第二に，Flash Crowds の対象となるコンテンツは局所的であり，更に継続時間も短い．すなわち，Flash Crowds 対策として負荷軽減するには，その発生している間に対象となっているコンテンツについてのみキャッシュが用意されていれば十分である．Aggressive Caching ではキャッシュを短期間で破棄することで，キャッシュとして価値の低いリクエスト頻度の低いコンテンツを共有ネットワーク上から速やかに排除する．これにより，いわゆるロングテールと呼ばれる大多数の不人気コンテンツのキャッシュに資源を消費することを回避する．

3.1.3 Query Origin Key

Query Origin Key は MashCache で用いる DHT において，キャッシュを管理するためのユニークなキーである．Query Origin Key はウェブサービスからコンテンツ取得の際に用いる HTTP クエリを基に生成する．本論文において HTTP クエリとは，HTTP リクエストに含まれる，コンテンツの Uniform Resource Identifier

(URI) や GET，POST メソッドなどで送信されるパラメータ，Cookie の値などを指す．たとえば，ブラウザ固有の情報である User-Agent やリクエスト元を示す Referer などは一般的にコンテンツを特定するための情報ではないため除外した上で Query Origin Key を生成する．あるコンテンツを取得しようとしているクライアントはその取得に必要な HTTP クエリを知っているため，キーとキャッシュの対応を管理するインデックスサーバなどを別途用意する必要はない．インデックスサーバが不要であることは，負荷集中や単一障害点が生じることを防ぐことができるため有益である．また，第 1.3.1 節で述べたように Flash Crowds 発生時のリクエストは一部のコンテンツに集中するため，HTTP クエリをキーとして用いることで，同等のキャッシュに紐付くキーが多数生成されることを防ぐことができる．

しかし，クエリはしばしばプライベートな情報を含むため，HTTP クエリをそのまま用いることはできない．MashCache ではこの問題を避けるために，HTTP クエリに対してハッシュ関数を適用することで得られる値を Query Origin Key として用いる．この方法により，インデックスサーバなどを用意せずにクライアントがキーを生成できるという Query Origin Key の基本設計を保ちつつ，HTTP クエリの秘密を守ることができる．また，ハッシュ値を用いることで，HTTP クエリが様々なフォーマットで含む各種値を DHT での利用に適した固定長のキーに変換することができる．

本研究において実装した MashCache のプロトタイプでは，Query Origin Key を生成するために用いるハッシュ関数における衝突は想定しない．すなわち，異なるコンテンツを取得するための HTTP クエリは常に異なり，また，異なる HTTP クエリから生成される Query Origin Key も常に異なるものとする．しかし，万が一ハッシュの衝突が発生した場合，次の 2 つのいずれかの方法でこの問題に対処することが可能である．第一の方法では，MashCache の共有ネットワーク上にキャッシュを配置する際に各キャッシュを元の HTTP クエリを暗号鍵として暗号化する．このようにすることで，異なる HTTP クエリから同一の Query Origin Key が生成された場合でも，一方の HTTP クエリでは復号できないためキャッシュミスとなり，誤ったキャッシュを取得することはない．但し，キャッシュの暗号化および復号のために計算コストを要することに注意が必要である．第二の方法では，複数のハッシュ関数を用いることで衝突を回避する．異なる HTTP クエリに対してそれぞれ複数の異なるハッシュ関数を適用した場合，各ハッシュ値が衝突する可能性は単一のハッシュ関数における衝突の可能性よりも低くなる．すなわち，複数種のハッシュ値を結合したものを Query Origin Key として用いることで，ハッシュ

の衝突可能性を低減させることが可能である．但し，衝突可能性を効果的に低減するには，特性を理解した上でハッシュ関数を選定する必要がある．もし用いるハッシュ関数が適当でない場合，衝突可能性を低減できないことに注意が必要である．

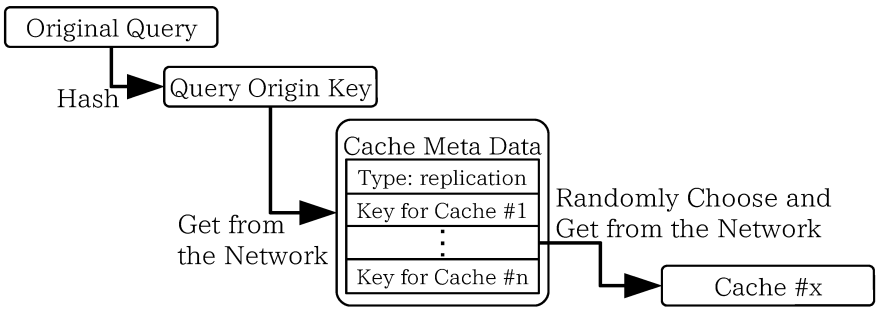
3.1.4 Cache Meta Data

Cache Meta Data (CMD) は MashCache で管理する各キャッシュにメタデータを付与し，キャッシュ管理の柔軟性を向上させる仕組みである．Cache Meta Data は，1) キャッシュ配信のための負荷分散および 2) キャッシュの保護に用いることができる．

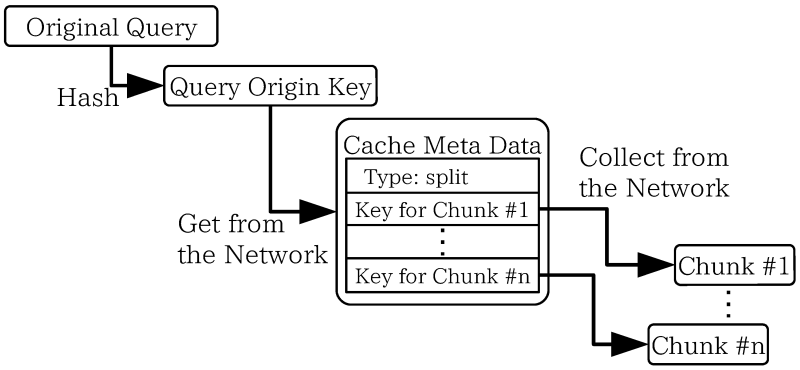
第一に，Query Origin Key では HTTP クエリとキャッシュを 1 対 1 で紐付けているため，Flash Crowds 発生時には人気のキャッシュを担当するクライアントに負荷が集中するという問題がある．しかし，Cache Meta Data を用いることによりひとつの HTTP クエリから複数のキャッシュにアクセスすることが可能となるため，キャッシュの分割や複製による負荷分散が可能となる．

図 3.3a に，キャッシュを複製することによる負荷分散の様子を示す．まず，クライアントはハッシュ関数を用い，目的のコンテンツを取得するための HTTP クエリより Query Origin Key を生成する．次に，生成した Query Origin Key を用い，共有ネットワークより Cache Meta Data を取得する．取得した Cache Meta Data には，キャッシュのレプリカが複数存在することを示す情報およびそれぞれを取得するためのキーが含まれている．クライアントは，これらのキーの中からランダムにひとつを選択し目的のキャッシュを取得する．このようにキャッシュを複製することにより，キャッシュを担当するクライアントがそれを配信する機会はレプリカの個数だけ分散される．

図 3.3b に，キャッシュを分割することによる負荷分散の様子を示す．分割した際には，Cache Meta Data に分割したという情報とともに各チャンク（キャッシュ断片）を取得するためのキーを記載する．Cache Meta Data を取得したクライアントは，これらのキーを用い必要なチャンクを取得した後，データの結合処理を行いキャッシュを取得する．このような分割は，たとえば，キャッシュのデータサイズが巨大な場合の負荷分散のために有効である．キャッシュのデータサイズが巨大な場合は，複製を生成して配信の機会を減らしたとしても各クライアントの負担は大きいためである．このような場合には，分割することにより各担当クライア



(a) 複製したキャッシュの参照



(b) 分割したキャッシュの参照

図 3.3: Cache Meta Data を用いたキャッシュの参照

ントの負荷を軽減させるとともに，異なるクライアントから複数のチャンクを並列で取得できるため取得に要する時間の短縮が期待できる．

第二に，MashCache の共有ネットワークで管理されるキャッシュは他のクライアントにも取得され得るという問題がある．すなわち，他のユーザによるキャッシュの中身の閲覧やキャッシュの改ざんなどの恐れがある．Cache Meta Data を用いることで，キャッシュの内容を保護することができる．たとえば，キャッシュを暗号化する場合などはその復号に必要な情報 (e.g., 暗号化アルゴリズム，復号鍵のヒントなど) を含むことで，必要なクライアントのみがキャッシュを復号可能なようにすることができる．このように Cache Meta Data を用いることで，無関係なクライアントによるキャッシュ取得を防ぐだけでなく，キャッシュの管理担当ノードが直接データの中身を読むことも防ぐことが可能となる．また，Cache Meta Data にキャッシュデータのハッシュ値を記載しておくことで，各クライアントが取得したキャッシュの整合性を確認することができる．これにより，キャッシュの破損や改ざんを検知することが可能となる．

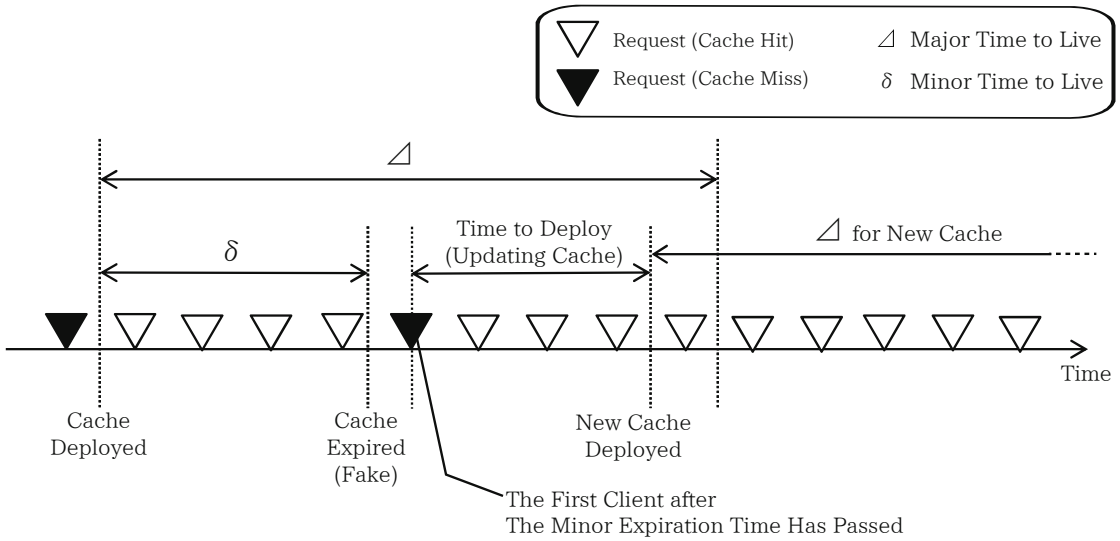
3.1.5 Two-phase Delta Consistency

Two-phase Delta Consistency は高頻度なキャッシュの更新とサーバの負荷軽減を両立するための技術である．高頻度なキャッシュの更新が必要であるのは，オリジナルコンテンツが更新された場合に早急にキャッシュに反映するためである．MashCache では，共有ネットワーク上に目的のコンテンツが存在しない場合 (キャッシュミス時) にのみ，ウェブサービスからオリジナルのコンテンツを直接取得するため，キャッシュを取得したクライアントはオリジナルコンテンツの状況を知ることはない．したがって，オリジナルコンテンツが更新された場合，その更新内容を早急にキャッシュに反映する必要がある．MashCache では，一定の短い周期でごく一部のクライアントに対して擬似的なキャッシュミスを発生させることで，ウェブサービスへのアクセス増加を抑えたままキャッシュを更新する．このようなキャッシュ管理を実現するために，Two-phase Delta Consistency では Minor Time-to-Live (Minor TTL) および Major Time-to-Live (Major TTL) として δ, Δ の 2 つの Time-to-Live (TTL) を定義する．Minor TTL は擬似的なキャッシュミスを発生させるまでの時間である．Major TTL は実際にキャッシュを破棄するまでの時間である．すなわち，δ と Δ は $\delta < \Delta$ の関係を持つ．

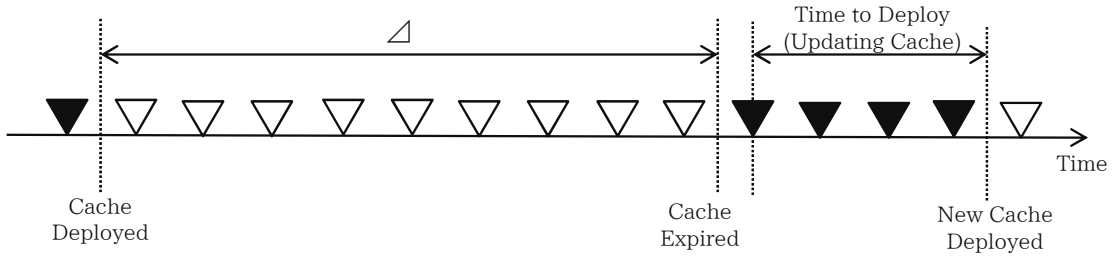
図 3.4 に，Two-phase Delta Consistency によるキャッシュ管理と単一の Time-to-Live によって管理する一般的な仕組みとの比較を示す．白い三角形および黒い三角形はそれぞれ，リクエストがキャッシュによって処理された場合 (キャッシュヒット) およびウェブサーバからオリジナルコンテンツを取得した場合 (キャッシュミス) を示す．この図に示すように Two-phase Delta Consistency は多数のクライアントがリクエストを発行している状況下に置いても，サーバへの負荷を増大させることなく高頻度なキャッシュ更新を実現する．

キャッシュの有効期限を TTL によって管理する一般的な仕組みの場合，キャッシュの有効期限が切れると次にキャッシュが配置されるまでサービスコアにクライアントのアクセスが集中してしまう．Flash Crowds 発生時においては多数のクライアントが殺到しているため，キャッシュの再配置までの僅かな時間においてもオリジンサーバに多数のアクセスが到達する．この問題を避けるために MashCache では，一般的な TTL を用いたキャッシュ管理を拡張した Two-phase Delta Consistency を用いる．

一方，Two-phase Delta Consistency では，キャッシュの更新を行う際に，サービスコアへクライアントのアクセスが殺到することを避けることができる．Minor



(a) Two-phase Delta Consistency によるキャッシュ更新



(b) Time-to-live (TTL) によるキャッシュ更新

図 3.4: キャッシュ更新方法の比較

TTL が切れた場合，担当クライアントはキャッシュの保持を継続するが，期限切れ後最初にアクセスしてきたクライアントに対してはキャッシュミスを返し，Minor TTL をリセットする．これによりキャッシュミスしたクライアントはサービスコアへ直接アクセスし，キャッシュの再配置の動作を開始する．再配置が完了するまでの間も，当該キャッシュは存在しているため後続のクライアントがサービスコアへ殺到することはない．一方，Major TTL が切れた場合は実際にキャッシュを破棄する．これは不要なキャッシュが長期間に渡って共有ネットワーク上に存在することを避けるためである．需要のあるコンテンツの場合は破棄される前に Minor TTL の作用により更新されるため，必要なキャッシュが破棄されることはない．

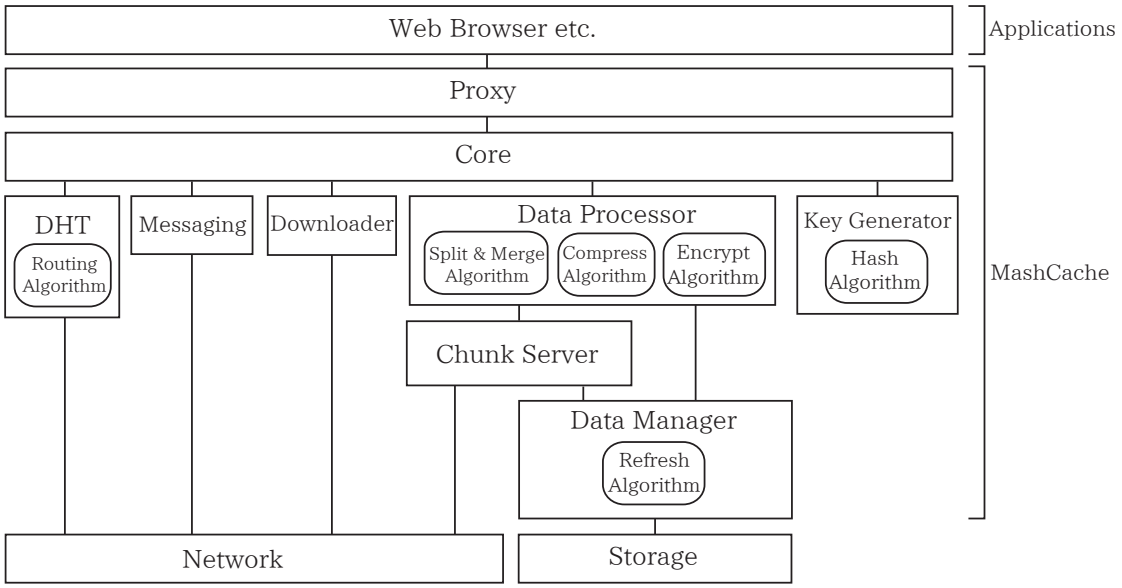


図 3.5: MashCache のアーキテクチャ

3.2 実装

本節では，MashCache のプロトタイプ実装について述べる．まず，アーキテクチャについて説明し，MashCache のプロトタイプの全体像と各コンポーネントの機能を説明する．

3.2.1 アーキテクチャ

MashCache は完全にクライアント側で動作する．また，ブラウザプラグインやローカル環境で動作するプロキシデーモンとして動作することを想定して設計している．図 3.5 に提案手法のアーキテクチャを示す．

Proxy は，ウェブブラウザなどのユーザアプリケーションからの HTTP リクエストを受け付け，また，MashCache が取得したキャッシュまたはオリジナルコンテンツをレスポンスとして返す．

Core は，1) DHT, 2) Messaging Service, 3) Downloader, 4) Data Processor, 5) Key Generator の 5 つのコンポーネントを制御する．

DHT, Messaging Service, Downloader の各コンポーネントは他のクライアントと相互にネットワーク接続する．DHT は，全てのクライアント，Cache Meta Data，キャッシュデータを管理する．本手法は DHT を用いて Cache Meta Data とキャッ

シュの探索を行う．この DHT に用いるルーティングアルゴリズムは，Flash Crowds 発生時のように膨大な数のクライアントが参加している状況でも正常に動作するよう，スケーラビリティの高いアルゴリズムである必要がある．本プロトタイプには，Overlay Weaver 0.9.9 [111–113] に含まれる Chord [114, 115] の実装を用いる．Overlay Weaver はオーバレイネットワーク構築ツールキットで DHT アルゴリズムの実装やシミュレーション環境を提供する．Chord はスケーラビリティの高い DHT アルゴリズムのひとつで，ノード数 N に対しノード探索に必要なホップ数は $O(\log N)$ で増加する．

Messaging Service は，クライアント間で授受されるメッセージを管理する．メッセージは，Cache Meta Data やキャッシュを配置または取得する際に用いられる．Downloader は，目的のキャッシュが存在しなかった場合に，ウェブサービスに直接アクセスしオリジナルコンテンツを取得する．すなわち，HTTP クライアントである．

Data Processor は取得したコンテンツまたはキャッシュの変換処理を行う．ウェブサービスからオリジナルコンテンツを取得した場合は，取得したコンテンツから Cache Meta Data を生成し必要に応じてキャッシュの加工も行う．一方，キャッシュを取得した場合は，Cache Meta Data の内容に基づいてキャッシュからオリジナルコンテンツを復元する．すなわち Data Processor は，分割および結合，圧縮および解凍，暗号化および復号といった対となる処理アルゴリズムを備える．

Key Generator は Cache Meta Data やキャッシュの管理に用いる Query Origin Key を生成する．Query Origin Key では集中管理することなくユニークなキーを取得するため，クエリおよび各データオブジェクトをハッシュ関数に与えることでそれぞれキーを生成する．本プロトタイプではキーの生成に MD5 ハッシュアルゴリズム [116] を用いる．キーは，クエリおよびオリジナルコンテンツの復元に必要なすべてのデータオブジェクトに対して生成する．

Data Manager は，Core とは独立して動作するコンポーネントで，クライアントが保持を担当する Cache Meta Data やキャッシュの寿命を管理する．すなわち，各 Cache Meta Data やキャッシュについて Two-phase Delta Consistency における 2 つのパラメータ Minor TTL，Major TTL を保持し，その寿命 (Major TTL) が尽きるまで管理を継続する．

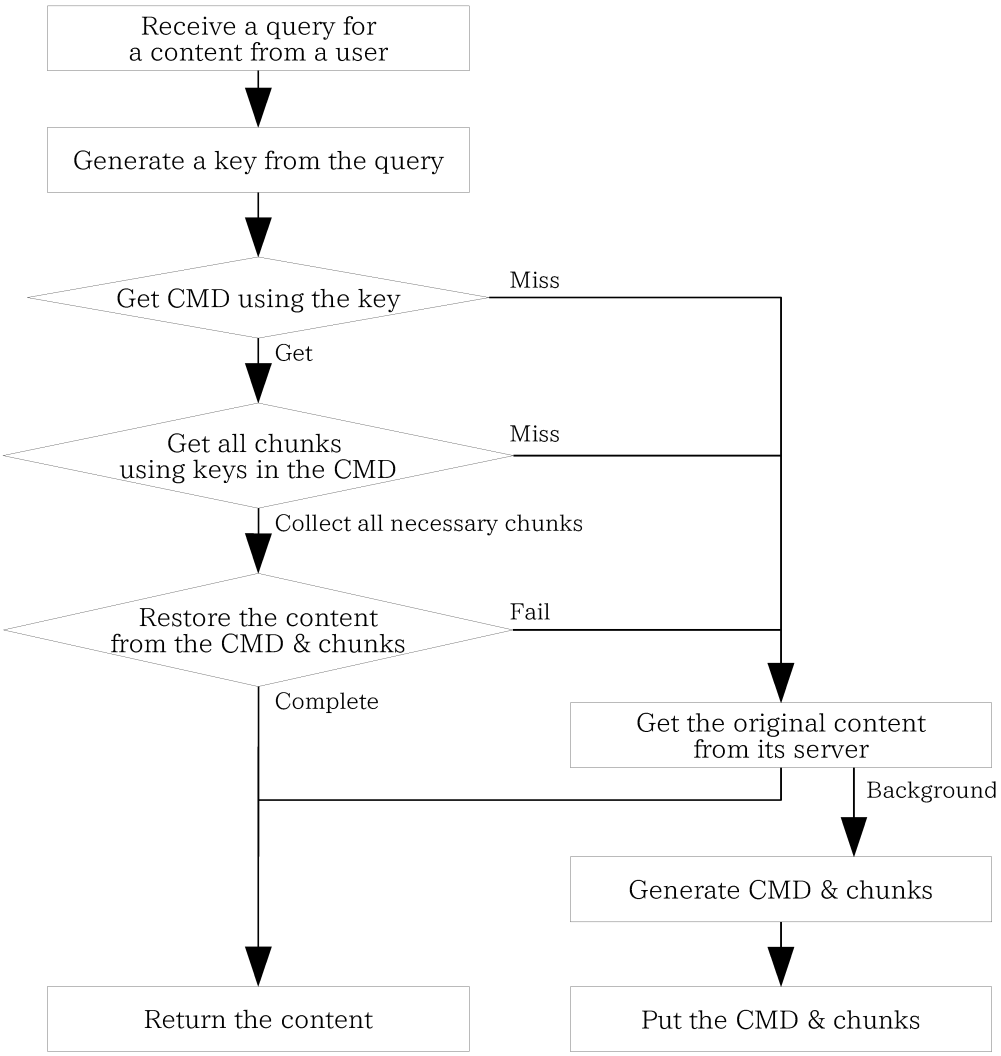


図 3.6: コンテンツ取得フロー

3.2.2 プロシージャ

本手法において，各クライアントはあらかじめキャッシュ共有用のオーバレイネットワークへ参加する．ブラウザやローカルプロキシデーモンなどの起動時にこのネットワークへ加入を行う．新たにネットワークに加入するノードは，既にネットワークに参加しているクライアントのひとつをブートストラップノードとして選出し，加入の手続きを行う．加入時のプロシージャは使用する DHT アルゴリズムに依存する．この共有ネットワーク上において各クライアントは Query Origin Key を用いて各キャッシュを管理する．

図 3.6 にクライアントがコンテンツを取得するまでのフローを示す．ウェブブラ

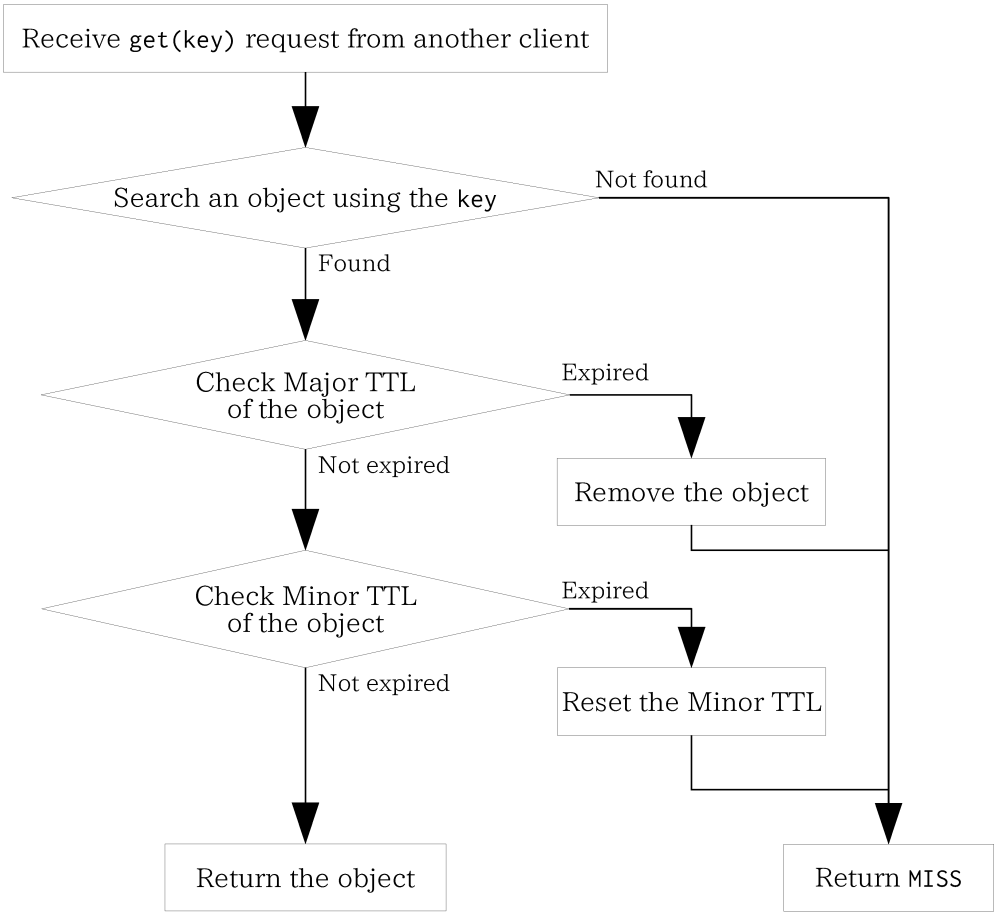


図 3.7: キャッシュ取得フロー

ウザなどのウェブサービスを利用するユーザアプリケーションがコンテンツを取得するためのリクエストを発行したとき，リクエストはウェブサービスに送信せず Proxy でフックする．Key Generator はフックしたリクエストをハッシュ関数に通し，Query Origin Key を生成する．その後，DHT を用いて，生成したキーに紐づく Cache Meta Data (CMD) を取得する．Cache Meta Data を取得後は Cache Meta Data に含まれるキーを用いてコンテンツの復元に必要なキャッシュを収集し，コンテンツを復元する．この一連の流れのいずれかで失敗が生じた場合はキャッシュミスとし，ウェブサービスよりオリジナルコンテンツの取得を行う．

キャッシュの取得

クライアント A があるデータオブジェクト（Cache Meta Data，チャンク，キャッシュのいずれか）の取得を試みる際，A はまず DHT を用いてそのキーを担当する

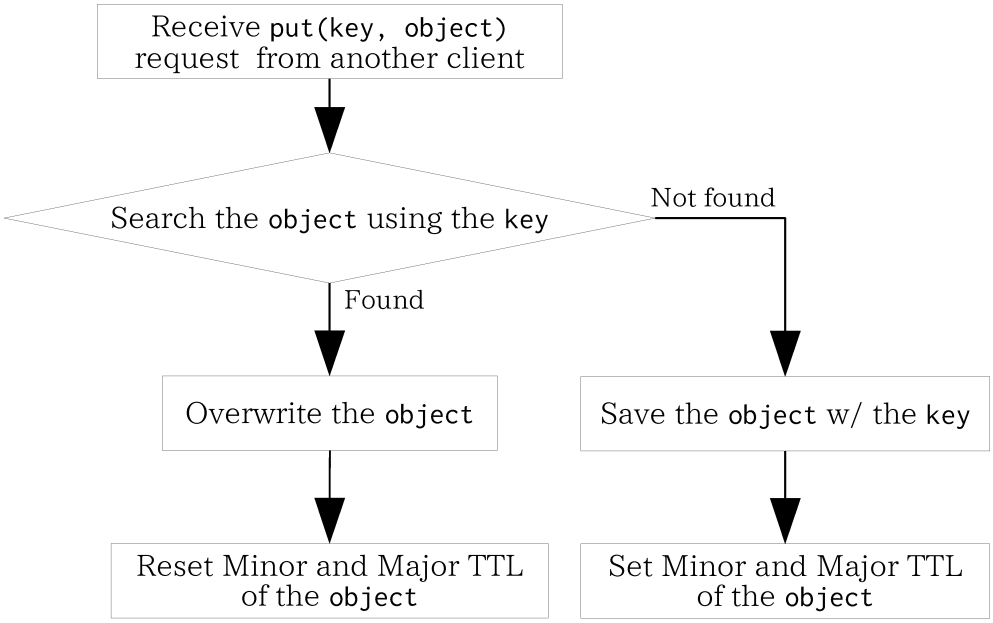


図 3.8: キャッシュ配置フロー

クライアント X を探索する．X を発見した後，A はキーとともに GET メッセージを X に送信する．図 3.7 に X が GET メッセージをどのように扱うかを示す．X はまず，ローカルストレージ内から受信したキーに該当するデータを探索する．該当するデータが見つからなかった場合はキャッシュミスを返す．一方，該当するデータが見つかった場合，Two-phase Delta Consistency に基づきそのデータの Minor TTL，Major TTL を確認する．Major TTL が失効している場合は，このデータを破棄しキャッシュミスを返す．Major TTL が有効な場合は，引き続き Minor TTL を確認する．Minor TTL が失効している場合は，データはそのまま保持し，Minor TTL のみ値をリセットする．Minor TTL が有効な場合は，キャッシュヒットとしてデータを返す．このようにデータの有効期限を管理することで，データを破棄することなく一定周期でキャッシュミスを引き起こし，キャッシュの更新を促す．

キャッシュの配置

一方，クライアント B があるデータを配置する際，B はまず DHT を用いてそのキーを担当するクライアント Y を探索する．Y を発見した後，B はキーおよびデータとともに PUT メッセージを Y に送信する．図 3.8 に Y が PUT メッセージをどのように扱うかを示す．Y はまず，ローカルストレージに対象のキーと紐づ

くデータが存在するかを確認する．該当するデータが存在する場合は，データを上書きし Minor TTL，Major TTL をリセットする．該当するデータが存在しない場合は，データを保存した上で，有効期限の管理情報を追加する．

3.3 評価

本節では，第 3.2 節で説明した提案手法のプロトタイプ実装を用い，MashCache が Flash Crowds の影響を軽減することを 3 つの実験によって示す．第一に，提案手法が Flash Crowds によるウェブサーバへの負荷を軽減することを示し，敏捷性が高いことを確認する．第二に，クライアント数が増加した場合におけるキャッシュの探索所要時間を示し，拡張性が高いことを確認する．最後に，提案手法がキャッシュの鮮度とサーバの負荷軽減を両立させることを示し，一貫性が高いことを確認する．なお，MashCache はクライアント資源のみを用いる設計により高い伸縮性を実現しているため，実験による評価は行わない．

本実験では最大で 2,500 台のクライアントをシミュレーションし MashCache を評価する．実験環境には表 3.1 に示す 2 台の物理マシンを用いた．本シミュレーションにおける各種パラメータの設定値を表 3.2 に示す．クライアント間の通信遅延は Meridian プロジェクト [117, 118] で使用されたデータセットを用い設定した．このデータセットは 2004 年 5 月 5–13 日の間に，実際のインターネット環境で稼働する 2,500 台の DNS サーバ間で計測された通信遅延 (Round Trip Time) を基にしている．各クライアントの帯域幅は 10 Mbps とした．ウェブサーバより配信するコンテンツのサイズは 1 MB とし，チャンクサイズの最大値は 256 KB とした．すなわち，各クライアントが取得したコンテンツは 4 分割された上で共有される．Two-phase Delta Consistency におけるデータの有効期限 Minor TTL，Major TTL はそれぞれ 5 s，10 s とした．また，リクエストが失敗時のリトライまでのインターバルは 1 s とした．

3.3.1 Flash Crowds による負荷の軽減

Flash Crowds によってウェブサーバの処理能力を超える負荷が発生した場合でも，MashCache は負荷を軽減し，ウェブサービスに障害が発生することを防ぐ．本項では，MashCache が Flash Crowds による負荷を軽減し，理想的な処理能力を備

表 3.1: MashCache の評価に用いる実験マシンの構成

	シミュレータ	ウェブサーバ
カーネル	Linux-2.6.20	Linux-2.6.31
CPU	Intel Xeon (Quad) 3.00 GHz	Intel Core2 (Quad) 2.40 GHz
メモリ	12 GB	2 GB
ネットワーク	Gigabit ethernet	
HTTP サーバ	-	Apache HTTP Server-2.2.12 [41]

表 3.2: MashCache の評価における各種パラメータの設定

パラメータ	設定値
クライアント間の通信遅延	75.8 ms （実環境における観測値）[117, 118]
各クライアントの帯域	10 Mbps
コンテンツサイズ	1 MB
チャンクサイズの上限	256 KB
キャッシュの有効期限	Minor TTL: 5 s, Major TTL: 10 s
リクエスト失敗時のリトライまでの時間	1 s

えたウェブサーバを準備した場合同様にすべてのリクエストを正常に処理できることを示す．

シミュレーション環境の制限により，本実験はワークロードをスケールダウンして行った．クライアント数を 100 台とし，その台数でも同時に接続しようとすると異常停止するようウェブサーバの帯域幅を 1 Mbps，最大同時接続数を 10 台と小さく設定した．この環境において，MashCache が Flash Crowds の影響を効果的に軽減していることを示すために，Direct と Oracle という 2 つの比較対象を設けた．Direct では，各クライアントが提案手法を用いず直接サーバへ接続し，コンテンツの取得を試みる．Oracle では，各クライアントの動作は Direct と同様だが，スケールダウン設定をしない．すなわち Oracle は，Flash Crowds 発生時においてもすべてのクライアントが直接サーバからオリジナルコンテンツを取得できる状況である．これは，Flash Crowds の規模や発生時期を正確に予測できる場合や，Flash Crowds の規模を無視できるほどのサーバ資源を常時しているような環

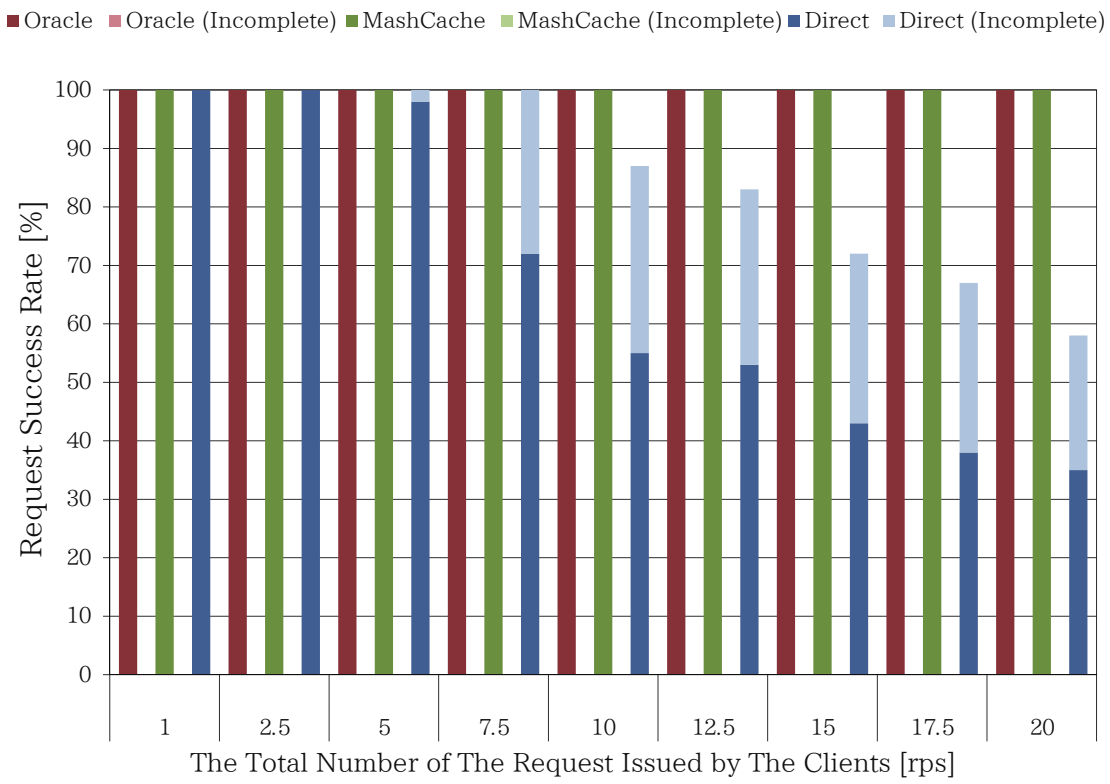


図 3.9: リクエスト発行頻度とリクエスト成功率の関係

境に相当するため，現実的な環境ではない．

本実験では，リクエスト成功率および毎秒のリクエスト処理数を計測した．リクエスト成功率 は各クライアントが発行したリクエストのうち実際にコンテンツを取得できた割合を示す．Incomplete はコンテンツの一部のみ取得できたことを示す．毎秒のリクエスト処理数 はウェブサーバが毎秒処理したリクエストの数を示す．Flash Crowds 発生時においてもこの値が低く保たれた場合，提案手法が Flash Crowds による負荷を軽減していることを示す．

リクエスト成功率

Flash Crowds の規模がリクエスト成功率に与える影響について評価した．図 3.9 にリクエスト発行頻度に対するリクエスト成功率の評価結果を示す．リクエスト発行頻度が大きいほど，Flash Crowds の規模が大きいことを示す．図の横軸は毎秒発行されるリクエスト数，縦軸はリクエスト成功率を表す．この結果より，Flash Crowds によって大量のリクエストが発行されている状況下においても，MashCache

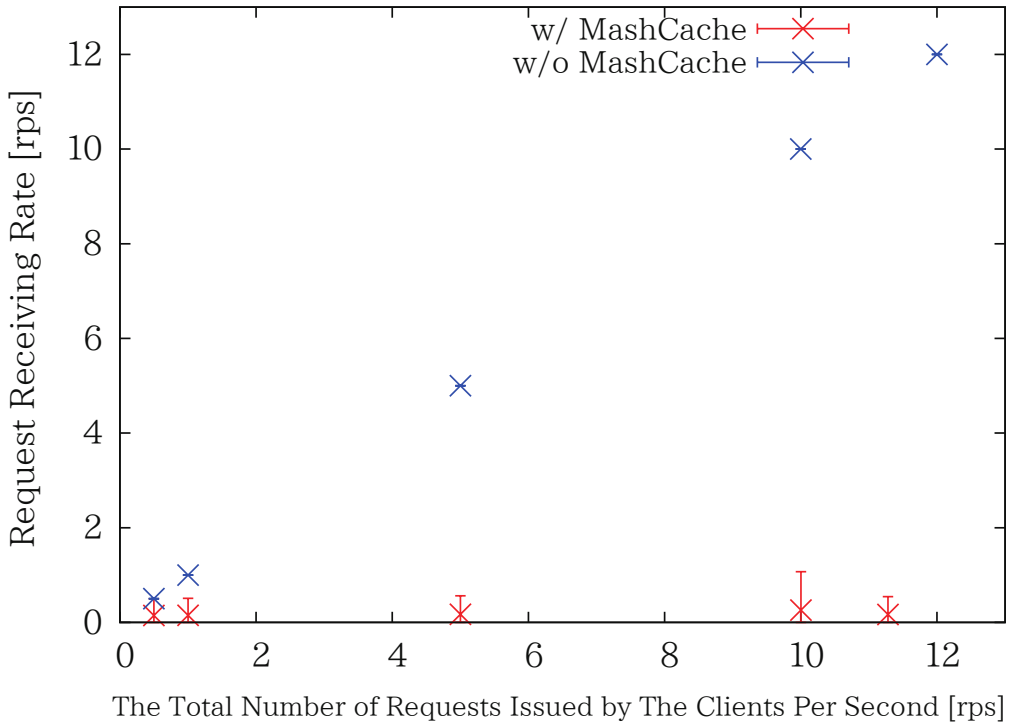


図 3.10: リクエスト発行頻度とリクエスト処理頻度の関係

を用いることで Oracle 同様に全てのリクエストが処理できたことがわかる．これは MashCache の共有キャッシュによって多くのリクエストが処理され，ウェブサーバに到達することがなかったためである．一方 Direct ではリクエスト頻度が毎秒 5 回の場合で 2 %，毎秒 20 回の場合で 65 % のリクエスト失敗が発生した．これにより，直接サーバがリクエスト処理をする状況では Flash Crowds によって容易にサービスの利用に悪影響が出ることがわかった．

サーバにおけるリクエスト処理頻度

MashCache は，クライアントのリクエストの大部分をサーバではなくクライアント間で処理することにより，ウェブサーバの負荷軽減を実現する．これを確認するために，Flash Crowds の規模とウェブサーバの負荷について，MashCache の有無による違いを比較した．図 3.10 に，毎秒のリクエスト発行数とウェブサーバが処理したリクエスト数の関係を示す．横軸はクライアントによって発行された毎秒のリクエスト数，縦軸はウェブサーバが処理した毎秒のリクエスト数を示す．エラーバーは標準偏差を示す．MashCache を用いない場合は，毎秒のリクエスト

発行数の増加に対してウェブサーバにおけるリクエスト処理数が線形に増加した．これはクライアントにより発行されたすべてのリクエストがウェブサーバで処理されているためである．一方，MashCache を用いた場合は，毎秒のリクエスト発行数に関係なく毎秒 0.2 回程度のリクエスト処理数であった．すなわち，毎秒 10 回のリクエストが発行された場合においては，発行されたリクエストの 98 % がクライアント間の共有キャッシュで処理されたことがわかる．毎秒のリクエスト発行数によらず，ウェブサーバにおける毎秒のリクエスト処理数が一定数であるのは Two-phase Delta Consistency における Minor TTL に基づきキャッシュミスが引き起こされウェブサーバへ直接リクエストが発行されるためである．本実験では，Minor TTL を 5 s に設定しているため，キャッシュミスの発生は 5 s に一度すなわち毎秒 0.2 回となり，評価結果とほぼ合致する．

3.3.2 スケーラビリティ

MashCache の拡張性が高いことを示すために，Flash Crowds の規模がコンテンツおよびキャッシュ取得の所要時間に与える影響を評価した．この評価では，Flash Crowds の規模として，毎秒のリクエスト発行数および共有ネットワークへの参加クライアント数の 2 つの指標を用いた．毎秒のリクエスト発行数は，対象のコンテンツを取得するためにクライアントによって発行される毎秒のリクエスト数を示す．参加クライアント数は，MashCache の DHT に加入しているクライアントの数を示す．理想的なシステムにおいては，毎秒のリクエスト発行数や参加クライアント数が増加しても共有ネットワークからキャッシュの取得に要する時間の増加は小さい．

毎秒のリクエスト発行数

MashCache は完全分散型の手法でクライアントとキャッシュの管理を行うため，多くのリクエストが発行される状況下においても，安定稼働する．これを示すため，毎秒のリクエスト発行数がキャッシュ取得時間に及ぼす影響について評価した．この実験では，100 台のクライアントを用いてシミュレーションした．

図 3.11 に，毎秒のリクエスト発行数が増加しても MashCache によってキャッシュ取得に要する時間の増加が抑えられることを示す．横軸は毎秒のリクエスト

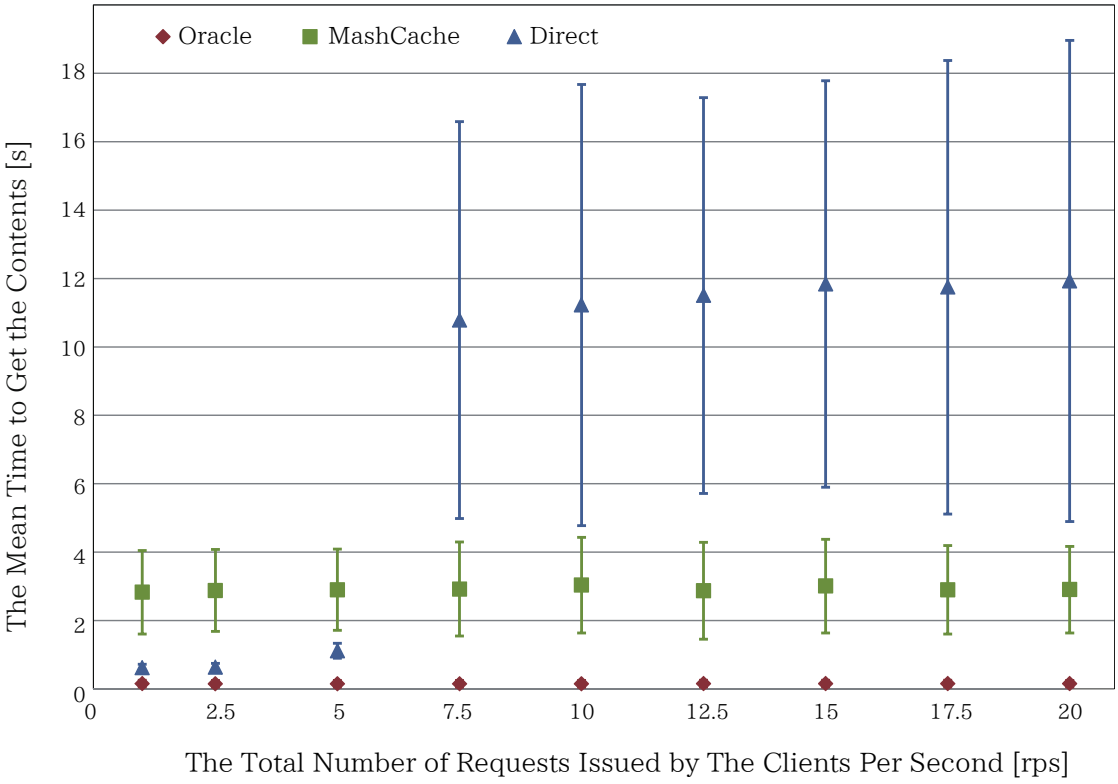


図 3.11: 毎秒のリクエスト発行数とキャッシュ取得に要する時間の関係

発行数，縦軸は共有ネットワークよりキャッシュ取得に要した時間を示す．エラーバーは標準偏差を示す．

毎秒のリクエスト発行数が増加した場合でも，MashCache と Oracle のコンテンツ取得時間は増加せず，それぞれ 2,915 ms ，152 ms で取得できた．一方 Direct ではリクエスト頻度が毎秒 7.5 回を超えた時にコンテンツ取得時間が 1,116 ms から 10,786 ms に増大した．これは負荷がウェブサーバの処理能力を超え多くのクライアントがコンテンツ取得に失敗したためである．コンテンツの取得に失敗したクライアントは 1 s 経過後にリトライするため，そのリクエストがウェブサーバの混雑をさらに大きくした．また， Direct においてはコンテンツを 1 度目のリクエストで取得できたクライアントとそうでないクライアントの差が広がり，標準偏差が拡大した．

また，この結果は提案手法のオーバヘッドも示している．本実験において MashCache によるコンテンツ取得時間の平均値は Oracle のそれよりも 2,764 ms 大きかった．詳細な解析により，このオーバヘッドの多くは DHT における探索に起因することがわかった．本実験で用いたプロトタイプは探索アルゴリズムに Chord

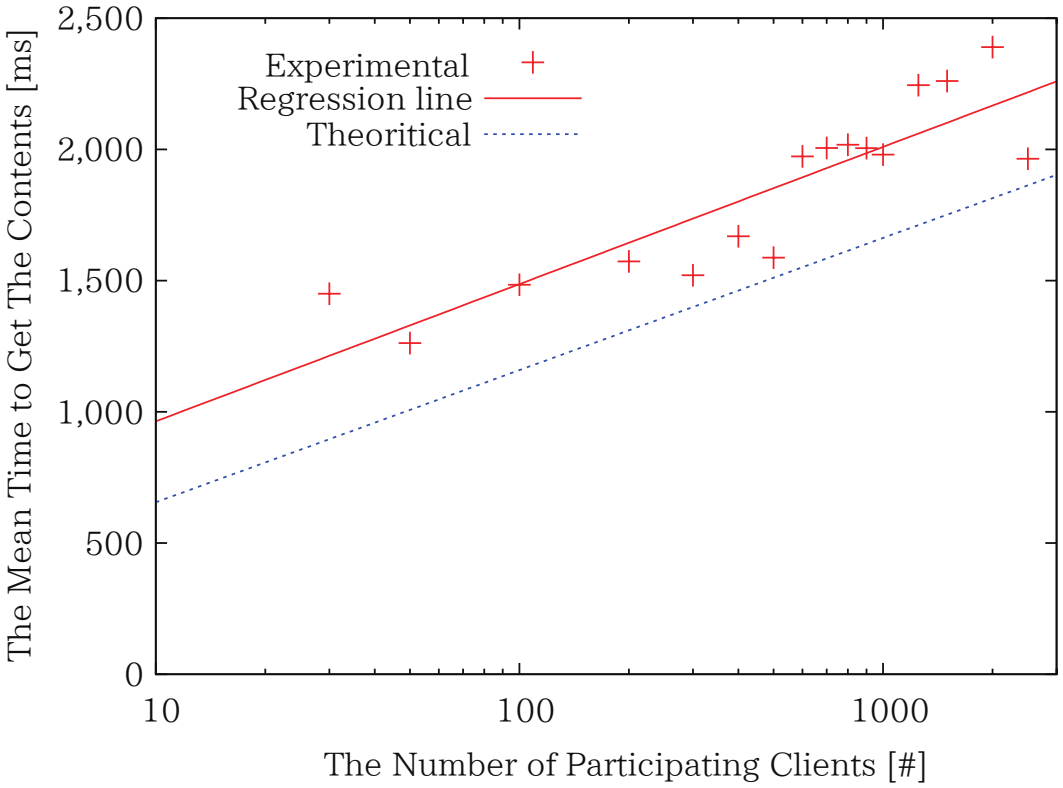


図 3.12: クライアント数とコンテンツ取得時間の関係

を用いているが，Chord は探索時間が大きいとの評価を受けている [119]．このようなオーバヘッドは，通信遅延を考慮したアルゴリズム [119,120] や地理的位置を考慮した複製配置手法 [121] などを用いることにより改善が見込める．

クライアント数

Flash Crowds 発生中は多数のクライアントが存在するため，Flash Crowds 対策手法においてクライアント数の増加に対するオーバヘッドの増加は小さい必要がある．本実験では，クライアント数が増加がコンテンツの取得時間に及ぼす影響について評価した．共有ネットワークに参加するクライアント数は 10 台から 2,500 台まで変化させた．

理論的な限界値を明らかにするために，まず，次式により MashCache において共有ネットワークからキャッシュ取得に要する時間の理論値を算出した．なお，本式においては，通信遅延と転送時間のみ考慮しており，取得後の Cache Meta Data やチャンクの処理時間については考慮していない．

表 3.3: 記号の定義

記号	説明
$\bar{t}_{\mathrm{cache}}$	クライアントがリクエストを発行してからキャッシュを取得するまでに要する時間
$\bar{t}_{\mathrm{search}}$	共有ネットワークにてあるキーの担当ノードの探索に要する時間
$\bar{t}_{\mathrm{cmd}}$	クライアントが Cache Meta Data の取得に要する時間
$\bar{t}_{\mathrm{chunk}}$	クライアントがチャンクの取得に要する時間
n	クライアント数
$\bar{l}$	Meridian データセットにおける通信遅延の平均値
v	通信路の帯域幅
$\bar{V}_{\mathrm{content}}$	オリジナルコンテンツのデータサイズ
$\bar{V}_{\mathrm{cmd}}$	Cache Meta Data のデータサイズ
$\bar{V}_{\mathrm{chunk}}$	チャンクのデータサイズ

$$\bar{t}_{\mathrm{search}} = \bar{l}\log n \tag{3.1}$$

$$\bar{t}_{\mathrm{cmd}} = \bar{t}_{\mathrm{search}} + \bar{l} + \frac{V_{\mathrm{cmd}}}{v} \tag{3.2}$$

$$\bar{t}_{\mathrm{chunk}} = \bar{t}_{\mathrm{search}} + \bar{l} + \frac{V_{\mathrm{chunk}}}{v}{\left\lceil \frac{V_{\mathrm{content}}}{V_{\mathrm{chunk}}} \right\rceil} \tag{3.3}$$

$$\bar{t}_{\mathrm{cache}} = \bar{t}_{\mathrm{cmd}} + \bar{t}_{\mathrm{chunk}} \tag{3.4}$$

式中の記号の定義を表 3.3 に示す．

図 3.12 にクライアント数とキャッシュ取得時間の関係を示す．横軸は共有ネットワークに参加するクライアント数，縦軸は共有ネットワークからキャッシュの取得に要した時間を示す．なお，横軸は対数軸である．Theoretical は $\bar{t}_{\mathrm{cache}}$ を示し，これはクライアント数 n に対し $O(\log n)$ で増加する．このような増加傾向を示すことは，ノード数 N の Chord ネットワークにおいて，目的のキーの探索に要するホップ数の増加が $O(\log N)$ であることに起因する．Experimental はプロトタイプを用いた実験結果を示す．計測値のプロットには最小二乗法による近似直線を添えている．この近似直線と理論値の R2 決定係数は 0.765 であり，プロトタイプによる結果もおよそ $O(\log n)$ の傾向を示すことがわかった．回帰値は理論値より約 440 ms 大きかった．この差は，Cache Meta Data やチャンクの処理などのオーバ

ヘッドに起因し，また，すべてのクライアントの処理を同一の物理マシン上で行うシミュレーション環境であるため大きな値になったと考えられる．また，この実験における解析は，クライアント数の増加に対するキャッシュ取得時間の増加はDHT における探索が MashCache のオーバヘッドとして支配的であることを明らかにした．本項で既に述べたように，DHT における探索時間を軽減させる手法を用いることで改善が見込める．

3.3.3 キャッシュの一貫性

負荷軽減手法は一般的にサービスコアが配信するコンテンツのキャッシュを用いるため，大規模なシステムでは特にオリジナルコンテンツとキャッシュの一貫性保持の方法が課題となる．本項では，MashCache が Aggressive Caching および Two-phase Delta Consistency によって，Flash Crowds 発生時でもキャッシュの一貫性を保ちつつウェブサーバの負荷を軽減できることを示す．そのために，次の 2 つの評価を行う．第一に，各クライアントが取得したキャッシュが生成からどのくらいの時間を経過しているかを評価する．この評価により，MashCache が頻繁にキャッシュを更新できていることを示す．第二に，ウェブサーバにおける毎秒のリクエスト処理数の推移を解析する．この評価により，MashCache がウェブサーバの負荷を増加させることなくキャッシュの更新を実現していることを示す．

本実験では，1) キャッシュ経過時間および 2) キャッシュ更新負荷を計測した．キャッシュ経過時間は，キャッシュが生成されてからの経過時間と定義する．キャッシュ経過時間が小さい場合，オリジナルコンテンツとキャッシュの間に差が生じ得る期間が短い．すなわちキャッシュの一貫性が高いことを示す．キャッシュ更新負荷は，キャッシュの更新に伴って発生するウェブサーバに対する負荷と定義する．キャッシュの更新負荷が小さいシステムであれば，キャッシュの有効期限が切れた際にウェブサーバにおける毎秒のリクエスト処理数を低く保つことができる．すなわち，Flash Crowds を適切に抑制できていることを示す．

キャッシュ経過時間

各クライアントが新しいキャッシュを取得できるように，MashCache は共有キャッシュを高い頻度で更新する．同一のキャッシュを更新せずに長期間保持することは，ウェブサーバの負荷軽減の目的としては非常に効果的であるが，第 1.3.2 項で

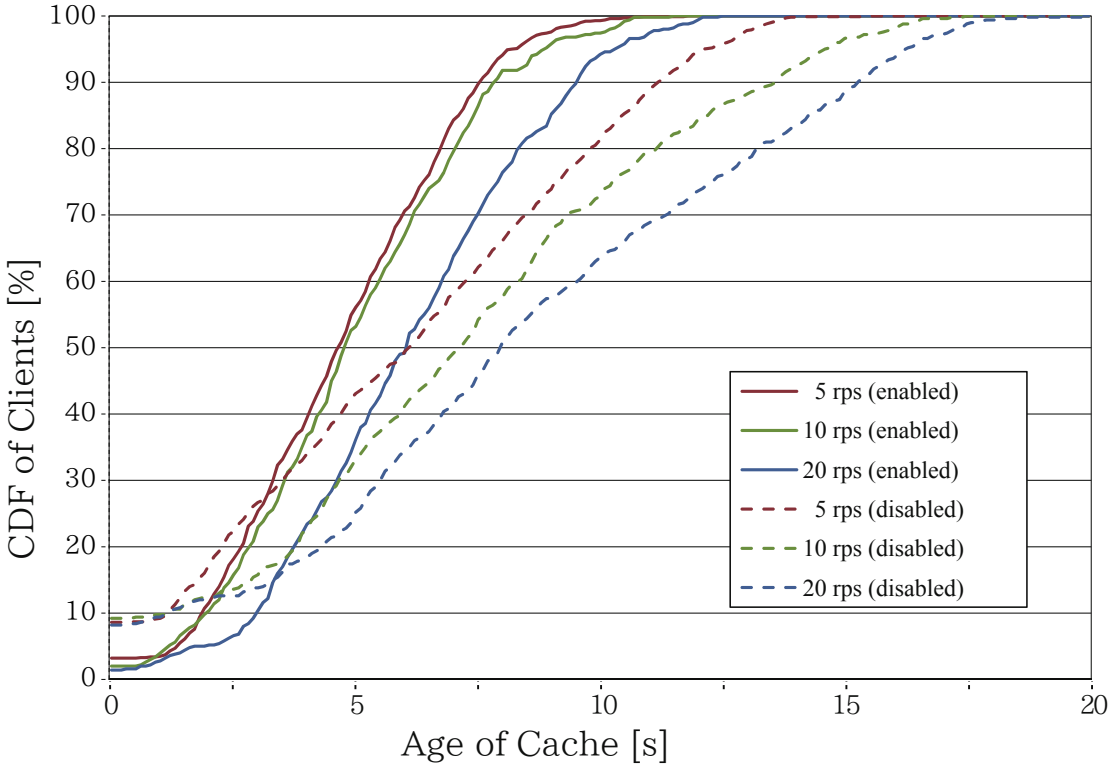


図 3.13: クライアントが取得したキャッシュ経過時間の累積分布関数

述べたように，キャッシュは高頻度で更新しオリジナルコンテンツとの一致を保つことが望ましい．MashCache がキャッシュ経過時間を低く保っていることを実験によって確認した．本実験では，Two-phase Delta Consistency のパラメータ Minor TTL，Major TTL は表 3.2 に示した通り，それぞれ 5 s および 10 s とした．また，Two-phase Delta Consistency を無効にしキャッシュの有効期限を 10 s に設定したものを比較対象とした．

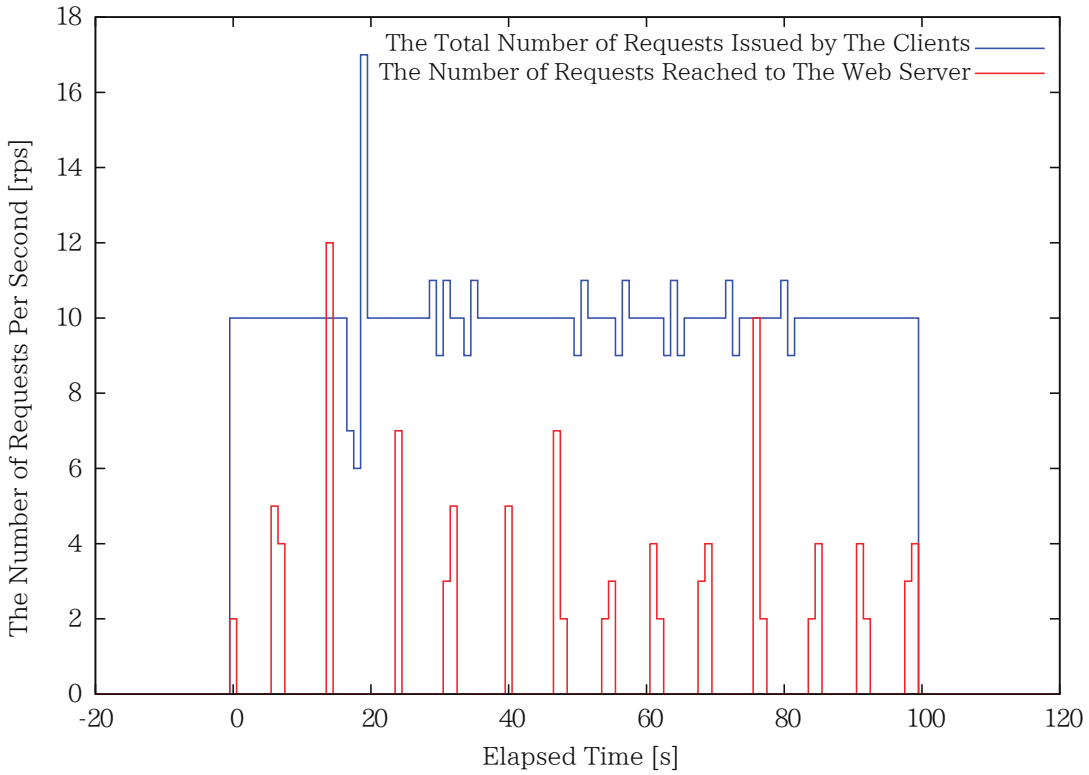
図 3.13 にクライアントが取得したキャッシュの経過時間の累積分布関数を示す．横軸はキャッシュの経過時間，縦軸はクライアントの累積分布関数を示す．ウェブサーバからオリジナルコンテンツを取得した場合はキャッシュ経過時間をゼロとする．各曲線は，毎秒のリクエスト発行数と Two-phase Delta Consistency の有無 (enabled または disabled) に対応する．この結果は，Two-phase Delta Consistency が毎秒のリクエスト発行頻度によらず，多くのクライアントが経過時間の小さいキャッシュを取得できていることが示す．たとえば，毎秒のリクエスト発行数が 10 回の場合，すべてのクライアントが取得したキャッシュの 90 パーセンタイル値は Two-phase Delta Consistency 有無の場合それぞれについて 7.9 s，13.6 s であり，

Two-phase Delta Consistency により各クライアントが新しいキャッシュを取得できることが明らかとなった．また，ウェブサーバから直接取得したクライアントの割合はそれぞれ 2.0 %，9.2 % であり，Two-phase Delta Consistency によってウェブサーバの負荷は低く保たれていることがわかった．

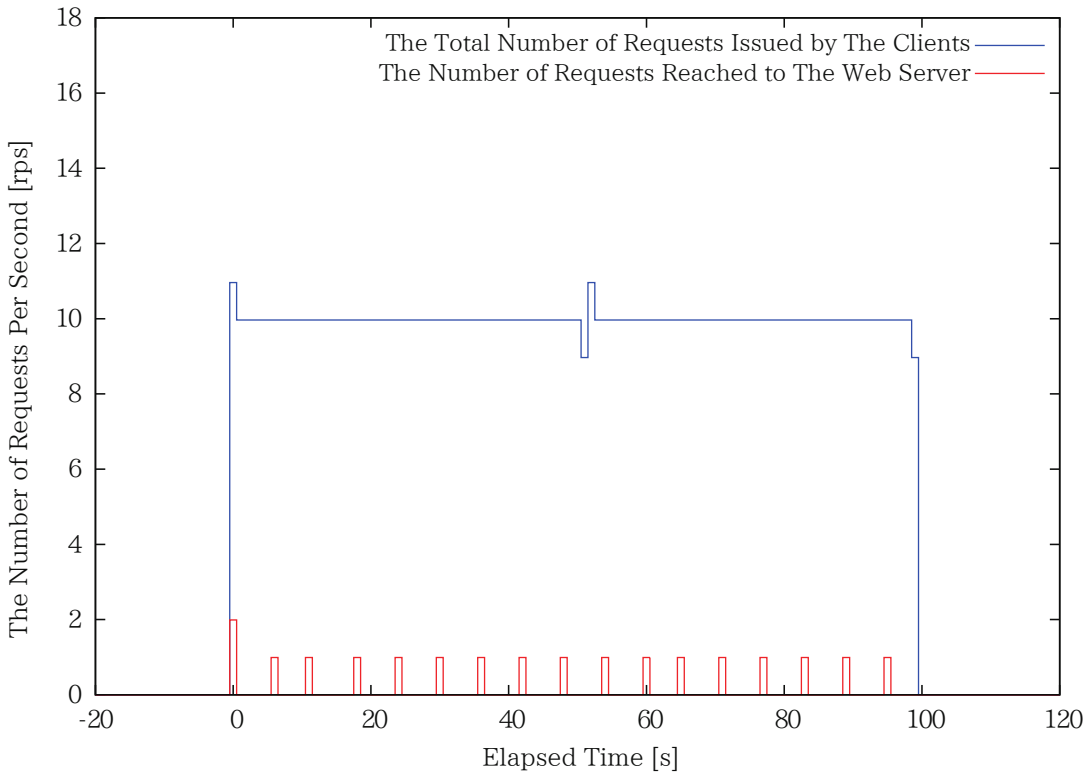
キャッシュ更新負荷

MashCache が，高頻度なキャッシュの更新を，ウェブサーバの負荷を増加させずに実現していることを示すために，ウェブサーバにおける毎秒のリクエスト処理数の推移を計測した．本実験では，Two-phase Delta Consistency のパラメータ Minor TTL，Major TTL は表 3.2 に示した通り，それぞれ 5 s および 10 s とした．また，Two-phase Delta Consistency を無効にしキャッシュの有効期限を 5 s に設定したものを比較対象とした．

図 3.14a，図 3.14b に，それぞれ Two-phase Delta Consistency を用いない場合と用いる場合の，ウェブサーバの負荷の推移を示す．横軸は時刻，縦軸は毎秒のリクエスト数を示す．この結果より，Two-phase Delta Consistency がウェブサーバの負荷を効果的に軽減していることがわかった．ウェブサーバにおける毎秒のリクエスト処理数の平均値は，Two-phase Delta Consistency の有無それぞれで 1 回および 10 回であった．この差異は，キャッシュの有効期限が切れた際にウェブサーバへリクエストが殺到する現象を Two-phase Delta Consistency が効果的に防いでいることによる．Two-phase Delta Consistency を用いない場合は，図 3.14a に示すように，キャッシュの有効期限が切れる度にウェブサーバへのリクエスト数が急増していることがわかる．一方 Two-phase Delta Consistency を用いる場合は，図 3.14b に示すように，ウェブサーバへのリクエスト数は小さい値で安定している．これは Minor TTL によるキャッシュミスが発生してキャッシュが再配置されるまでの間も，元のキャッシュが利用可能なためである．



(a) Two-phase Delta Consistency を用いない場合



(b) Two-phase Delta Consistency を用いる場合

図 3.14: ウェブサーバにおける毎秒のリクエスト処理数の推移

表 3.4: ネットワーク帯域消費量の見積もりに用いる記号の定義

記号	説明
$\bar{c}$	各クライアントが担当するキャッシュの数の平均値
$\bar{V}_{\text{content}}$	オリジナルコンテンツのデータサイズ
V_{chunk}	チャンクのデータサイズ
n	クライアント数
u	クライアントが同時に発行するユニークなクエリの数
$\bar{v}$	各クライアントの通信路のアップロード帯域幅の平均値
$\bar{t_c}$	チャンクの転送に要する時間の平均値
T	各キャッシュの有効期限 (time-to-live)

3.4 議論

3.4.1 オーバヘッド

ネットワーク帯域消費量

MashCache において，ネットワーク帯域消費量は 2 つの理由で増加する．第一の理由は，DHT を用いた探索や DHT の構築および保持のための通信である．この通信によるネットワーク帯域消費量は無視できるほど小さいと考えられる．なぜならば，多くのコンテンツデータが数 KB より大きいことに対し，これらのデータ量は数十 B 程度であるためである．第二の理由は，キャッシュを担当のクライアントに保存するためのデータ転送である．あるクライアントが担当のキャッシュを受け取るための帯域消費量 E は次式で見積もることができる．式中の各記号の定義は 表 3.4 に示す．すなわち，MashCache 全体におけるネットワーク帯域消費量は nE で見積もることができる．

$$\bar{c} = \frac{\bar{V}_{\text{content}}}{V_{\text{chunk}}} \cdot \frac{u}{n} \tag{3.5}$$

$$\bar{t_c} = \frac{V_{\text{chunk}}}{\bar{v}} \tag{3.6}$$

$$E = \frac{u}{n} \sum_{i}^{\bar{c}} {}_{\bar{c}}C_i i \left(\frac{\bar{t_c}}{T}\right)^i \left(1 - \frac{\bar{t_c}}{T}\right)^{\bar{c}-i} \tag{3.7}$$

具体例として，$n = 10^6$, $\bar{v}$ = 1 Mbps, T = 5 s, V_{chunk} = 256 KB, $\bar{V}_{\text{content}}$ = 1 MB という状況を考える．この状況において，各クライアントがすべて異なるコンテンツ

をリクエストした場合 ($u = n$)，各クライアントにおけるネットワーク帯域消費量 E は 0.8 Mbps となり，全体では nE = 800 Gbps となる．この全体のネットワーク帯域消費量は 2010 年のインターネット全体の毎秒のトラフィックの約 1 % に相当する [122].

この例では，すべてのクライアントが完全に異なるリクエストを同時に発行する計算を行っているため，最も悪い場合の値である．しかし，このように，すべてのクライアントが完全に異なるリクエストを発行することは稀な状況であり，実際に MashCache によるネットワーク消費量はより小さくなると考えられる．なぜならば，一般的に，ウェブコンテンツの人気は Zipf 分布となることが示されており，一部のコンテンツにリクエストが集中するためである [123]．同じコンテンツへのリクエストが増加することは u が小さくなることを示しており，MashCache によるネットワーク帯域消費量も小さくなる．

応答遅延

DHT は目的のキャッシュを保持するノードの探索に数回のホップを必要とする．つまり，ウェブサーバへ直接リクエストを発行する場合に比べ，MashCache の共有ネットワークからキャッシュを取得する場合は，探索に伴う通信遅延の分リクエストの応答時間が大きくなる．Flash Crowds が発生していない場合，MashCache の各クライアントはウェブサーバの応答時間に加え $\bar{t}_{\mathrm{search}}$ を要する．なぜならば，Cache Meta Data を探索しキャッシュミスとなった後に，ウェブサーバから直接オリジナルコンテンツを取得するためである．各クライアント間の通信遅延を 75.8 ms [117,118] とした場合，クライアント数 $n \approx 9.35 \times 10^3$ のときに探索時間 $\bar{t}_{\mathrm{search}}$ は 1 s となる．同様に，$n \approx 9.04 \times 10^5$ のとき 1.5 s，$n \approx 8.76 \times 10^7$ のとき 2 s となる．したがって，MashCache における応答遅延の増加は 1 s から 2 s 程度であると考えられる．また，通信遅延を考慮した DHT [120] などを利用することで，この応答遅延は 250 ms 程度まで減少させることができる．

すなわち，参照の深さが 3 以下の場合，Flash Crowds が発生していない状況における応答遅延の増加は 1 s 以下に収まると考えられる．ここで参照とは，ウェブコンテンツが別のコンテンツを参照していることを意味する．たとえば，リクエストしたある HTML ページ `foo.html` がスタイルシート `bar.css` とスクリプト `baz.js` を参照しており，さらに `bar.css` が画像 `qux.jpg` を参照している場合， `qux.jpg` の深さを 3 と定義する．MashCache は全体の応答遅延の増加を抑

表 3.5: コンテンツのデータサイズと応答遅延の影響に関する試算

Percentage of Latency [%]	Latency [ms]				
	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

制するために，可能な限り並列にキャッシュの取得を試みる．上記の構成例の場合，`bar.css` および `baz.js` は同時に取得を行う．すなわち，参照の深さを d とする場合，最も参照の深いコンテンツを取得するまでのオーバヘッドは $d\bar{t}_{\mathrm{search}} = d\bar{l}\log n$ と見積もることができる．通信遅延を考慮した DHT を用い $\bar{t}_{\mathrm{search}}$ = 250 ms の場合，750 ms のオーバヘッドとなる．

MashCache はデータサイズの大きいコンテンツに対してよりよい効果を得ることができる．なぜなら，データの転送時間に対し応答遅延が相対的に小さくなるためである．MashCache において，数 MB のデータを取得する場合，応答遅延の占める割合はおよそ 10 % 以下である．表 3.5 に，コンテンツ取得時間対し応答遅延の占める割合およびコンテンツのデータサイズの関係について試算を示す．本試算において，各クライアントのネットワーク帯域幅は 5 Mbps としている．各クライアントのネットワーク帯域幅を 5 Mbps としたのは，インターネットユーザの 78 % が 5 Mbps 以下の通信路を利用しているとの統計データ [124] に基づく．本試算によると，たとえば，応答遅延が 250 ms の環境において 1.41 MB のコンテンツを取得する場合の応答遅延は約 10 % である．また，同様の環境において 15.5 MB のコンテンツを取得する場合の応答遅延は約 1 % である．すなわち，データサイズが大きく転送時間が増加するほど，応答遅延は相対的に小さくなる．同様に，ネットワーク帯域幅が小さいほどデータの転送時間が増加するため，この場合も応答遅延は相対的に小さくなる．

オーバヘッドの削減

MashCache は Aggressive Caching に伴いネットワーク帯域を消費するが，応答

遅延についてはキャッシュの利用を適切に選択することによって削減することが可能である．Aggressive Caching において各クライアントは，キャッシュが存在しない場合や有効期限が切れている場合に新たにキャッシュを生成するが，これはキャッシュの利用を必ずしも強制するものではない．Flash Crowds が発生していないときは，サービスコアへ直接リクエストを発行し，同時にキャッシュの有無を確認することでキャッシュミスによるオーバヘッドの増大を抑制することができる．この場合，サービスコアからの応答が極端に遅い場合などは Flash Crowds が発生していると見なし，それ以降そのサービスへのリクエストは共有キャッシュを優先するように切り替えるなどの仕組みが必要となる．

3.4.2 セキュリティ

プライバシ

各クライアントが生成したキャッシュのうち秘匿性の高いものは暗号化によって保護する必要がある．MashCache では，1) オリジナルのクエリを用いてキャッシュを暗号化する，2) サービス提供者よりキャッシュ禁止指定のコンテンツについてはプライベートキーを用いて暗号化する，3) サービス提供者よりキャッシュ禁止指定のコンテンツは MashCache の共有ネットワークにアップロードしない，という 3 つの拡張により秘匿性の高いコンテンツを保護することが可能である．

第一の拡張では，各クライアントが取得したコンテンツをキャッシュとして共有する前に，コンテンツの取得に用いたオリジナルのクエリを用いて暗号化を行う．もしキャッシュがプライベートな情報を含む場合，クエリはそのクライアント固有の情報を Cookie やパラメータなどに含む．そのため，キャッシュされるべきでないコンテンツの多くは，他のクライアントからは復号不可能となり，プライバシは保護される．

第二の拡張では，サービス提供者よりキャッシュ禁止との指定のあるコンテンツに関しては各クライアントがプライベートキーを用いて共有するキャッシュを暗号化する．キャッシュ禁止の指定とはたとえば，HTTP レスポンスや HTML ヘッダにおいて `Cache-Control` が `private` や `no-store` と指定してある場合である．この拡張では，クライアントの IP アドレスによるアクセス制御が行われている場合などにも適用可能である．このようなアクセス制御が行われている場合，第一の拡張では同一のクエリを他のクライアントが発行することが可能であるため，

本来キャッシュを取得すべきでないクライアントがキャッシュを取得できてしまう問題がある．一方この第二の拡張ではそのような問題にも対応可能である．しかし，第二の拡張では，サービス提供者がキャッシュの可否を適切に指定してあることが前提である．秘匿性の高くないコンテンツにもキャッシュ禁止の指定がされている場合，MashCache における負荷軽減の効果が低減する．

第三の拡張では，サービス提供者より共有禁止との指定のあるコンテンツに関しては共有ネットワークへ一切アップロードしない．たとえば，第二の拡張と同様に，`Cache-Control` が `private` や `no-store` と指定している場合，共有ネットワークへのアップロードを行わない．この拡張は第二の拡張よりも強固にプライバシを保護する．しかし，第二の拡張と同様に，不適切な設定はプライベートな情報の流出や負荷軽減効果の低減に繋がる．第二，第三の拡張が抱えるこのような問題は，プロキシ方式のように広く使われている手法でも同様である．

第一の拡張と，第二，第三の拡張は補完的な関係にある．第一の方法は，アクセスクライアントの識別情報がリクエストに含まれている場合に効果的である．このような場合，プライベートなコンテンツは共有ネットワーク上に配置されるが暗号化により保護される．また，サービス提供者における特別の対応は必要としない．一方，第二，第三の拡張はクエリの内容によらずサービスコアからのレスポンスの指示に従ってキャッシュの暗号化や共有を中止する．但し，これらの拡張はサービス提供者が適切にキャッシュの可否を指定する必要がある．

キャッシュの完全性

MashCache はクライアント側の Peer-to-Peer ネットワークに依存するため，キャッシュの完全性を保証することは困難である．しかし，サービス提供者がキャッシュの改ざんを確認することは容易である．なぜなら，MashCache は DHT を用いた構造化オーバレイを用いており，保存されたキャッシュをサービス提供者が取得することは可能である．Aggressive Caching によって頻繁にキャッシュが更新されるため，改ざんしたキャッシュはすぐに新しいキャッシュで上書きされる．

3.4.3 キャッシュヒット率

MashCache においてキャッシュヒット率を向上させるには，ウェブページ単位ではなくオブジェクト単位でキャッシュを生成する必要がある．オブジェクトとは，

表 3.6: Amazon.co.jp のトップページを構成するオブジェクト

	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

ウェブページを構成する各要素を指す．MashCache は各オブジェクトを取得するためのクエリが異なる場合，異なるキャッシュとして共有ネットワークに配置される．したがって，ウェブサービスが配信するコンテンツの多くがプライベートな情報であるか否かに関わらず Cookie などの情報を要求している場合，たとえそればプライベートな情報を含まない場合でもキャッシュが共有されないため負荷軽減の効果を得られない．実際に Cookie が要求されるのはどのようなオブジェクトであるか，ショッピングサイト大手の amazon.co.jp のトップページにログイン済の状態でアクセスし，Cookie を必要とするオブジェクトの個数するか調査した．調査の結果，表 3.6 に示すように，Cookie を必要とするのはデータサイズで 5.8 % のオブジェクトであり，個人のページであっても画像をはじめとする多くのオブジェクトが Cookie を必要としないことがわかった．

3.4.4 一貫性

MashCache は次の 2 つの状況において，オリジナルコンテンツと異なる内容のキャッシュを返す可能性がある．第一に，キャッシュが生成された後にオリジナルコンテンツが変更された場合である．この状況に対応するため，第 3.1.5 項で述べたように，MashCache では Two-phase Delta Consistency によって，サービスコアの負荷を増加させることなく高頻度なキャッシュの更新を実現している．加えて，MashCache は更新と異なる順序でキャッシュを取得しないことを保証している．これは DHT によって特定される位置にキャッシュが保存されるためであり，キャッシュの更新を容易にしている．

第二に，コンテンツがサーバの状態に依存して生成される場合である．このような場合は，サービス提供者によってキャッシュ保存の可否を適切に指定されている必要がある．たとえば，あるリクエストの処理結果が時刻や乱数に依存して変化する場合など，クライアント側ではそのような挙動であるかどうかは分かり得ないため，サーバ側がキャッシュの可否を指定しなければならない．このような問題は，プロキシ方式のような既存手法と同様である．

3.5 まとめ

本章では，クライアント間連携によるサービスコアの負荷分散手法 MashCache について説明した．本手法は，クライアント側における完全分散型のアーキテクチャにより，効果的に Flash Crowds による負荷を軽減する．本手法の設計には，Flash Crowds の詳細な解析結果が用いられている．この解析に基づく，1) Aggressive Caching, 2) Query Origin Key, 3) Two-phase Delta Consistency, 4) Cache Meta Data などの仕組みにより，Flash Crowds 発生時における負荷分散と高頻度なキャッシュ更新を実現する．シミュレーションによる実験では，これらの各仕組みが効果的に作用していることを定量的に示し，提案手法の有用性を示した．

第4章　データセンタ間連携による ストレージの負荷分散手法

本章では，異なるデータセンタに属するストレージサーバ（ノード）の資源を統合し，複数のデータセンタに負荷を分散する手法 Pangaea について述べる．本手法は，分散ハッシュ表 (DHT) を用い，単一の巨大なキーバリューストアを構築し，複数のデータセンタを跨ぐストレージ層を構築することによってクラウド方式および完全複製方式における伸縮性の問題を解消する．また，データセンタ内通信路と比較して，高遅延かつ狭帯域であるデータセンタ間通信路を利用することに起因する問題を 2 つの要素技術 1) Multi-Layered Distributed Hash Table および 2) Local-first Data Rebuilding により解決する．本章では，まず，複数のデータセンタを利用するストレージの構築にあたり，キーバリューストアが有用であることと問題点について整理する．次に，Pangaea を構成する要素技術について述べ，どのように問題を解決するかを説明する．つづいて，実装について説明し，各要素技術の効果をシミュレーション環境および実際のインターネット環境を用いた実験により評価する．その後，Multi-Layered Distributed Hash Table のオーバヘッドについて議論し，最後に本章をまとめる．

4.1　データセンタ環境とキーバリューストア

第 2 章で述べたように，拡張性の高いウェブサービスの負荷分散手法として，クラウド方式や完全複製方式の運用が広く利用されているが，いずれの場合もデータセンタの制約を受ける．その一方で，ウェブサービスの社会的重要性の高まりを受け，単一のデータセンタではサービスの要求性能を満たさない場合があり，データセンタの制約を受けずに負荷分散することが求められている [105]．複数のデータセンタの資源を用いた理想的なクラウド環境では，各データセンタで稼働している資源を集約しひとつの巨大なクラウド環境を構築することで，単一のデータセンタで構築されたクラウド環境よりも柔軟な資源管理を実現する．たとえば，この

ように複数のデータセンタを使用したクラウド環境上では，運用されているサービスの利用状況等に応じて，使用するサーバインスタンスの台数，地理的な場所，稼働させる時期をより高い自由度で選択することが可能となる [125–127].

Flash Crowds 対策として負荷分散する場合も同様で，より多くの資源を利用するには，単一のデータセンタの制約を受けずウェブサービスを展開可能にする方法が必要である．このように膨大な資源を効率的に利用可能なストレージのひとつとして分散キーバリューストア [66–68, 70–76] がある．分散キーバリューストアは高い拡張性を実現すると共に，データの物理的な配置を抽象化して提供する．分散キーバリューストアは複数のストレージサーバ（ノード）から構成され，その台数を増加させることで容易に全体の容量や I/O スループットを向上させることが可能である．また，分散キーバリューストアを利用するアプリケーションは，データが物理的にどのノードに保存されているかという情報を知る必要は無い．

分散キーバリューストアはこれらの特徴を，図 4.1 に示す 4 層のソフトウェアスタックによって実現する．但し，手法によっては一部の層の機能が極小化されている場合もある．また，このような構造の整理には Overlay Weaver [113] におけるアーキテクチャを参考にした．キーバリューストア API (Application Programming Interface) 層は，アプリケーションがストレージを使用するための `put`, `get` などの汎用的な API を提供する．MondoDB [73] のように SQL に類似するクエリに対応するなど，高度な API を提供するものも存在する．データフェッチ層は，オリジナルのデータオブジェクトと保存時の形態 (チャンク) を相互に変換する．たとえば，対象データを複数の断片に分割した上で各断片を異なるノードに保存することで，各ノードの負荷を分散し I/O を並列化しスループットを向上させる．ルーティング層は，要求されたチャンクを実際に保持しているノードの探索を行う．各ノードが保持するチャンクの管理は，インデックスサーバを用いる集中管理型の手法と DHT などを用いる分散管理型の手法がある．ストレージ層は，保存担当のノードがチャンクをローカルに保存する．データの保存には，ディスクのみでなく主記憶装置を使用する場合もある．

一般的に分散キーバリューストアを構成する各ノードは，状態確認，被リクエストデータの探索，データの送受信などの通信を高頻度に行う．しかし既存の分散キーバリューストアにおいてこのような通信は，データセンタ内のように高速な通信路を介して行われることが想定されている．したがって，各ノードの間に高遅延かつ狭帯域であるデータセンタ間通信路が存在する場合，それを利用する分散キーバリューストアの通信遅延は増大しスループットは低下する．すなわち，

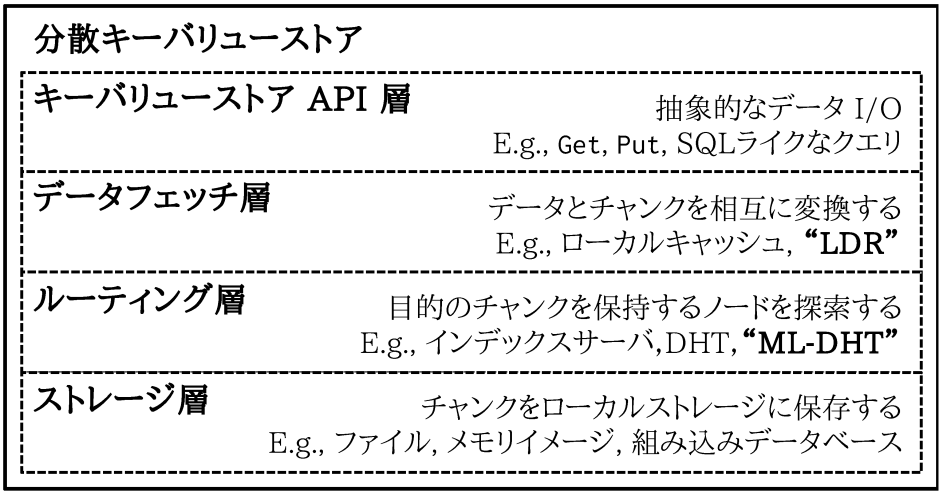


図 4.1: 分散キーバリューストアを構成するソフトウェアスタック

データセンタ間を跨いで構築した分散キーバリューストアは性能を十分に発揮することができず，これを利用するアプリケーションの応答性やスループットを顕著に低下させる．したがって，データセンタ間を跨ぐ分散キーバリューストアを構築するには，データセンタ間の通信頻度と転送量を抑えることが重要である．

4.2 設計上の課題

複数のデータセンタ間を跨ぐ分散キーバリューストアは，単一のデータセンタ内で運用される分散キーバリューストアに対して，伸縮性が高い．これは複数のデータセンタを統合した方が利用可能となる資源の量や選択肢が増加するためである．これは同時に，管理対象の資源が多いことも意味するため，スケーラビリティの観点から，本研究では DHT を用いる分散管理型の分散キーバリューストア [95,114,115,128–131] を対象とする．このような分散キーバリューストアでは，属するデータセンタに関わらず全てのノードが，互いに通信を行うことで目的のデータを探索することが可能である．

本節では複数のデータセンタを跨ぐ分散キーバリューストアの課題について，1) ストレージ性能の均等な拡張性，2) データセンタ間通信の頻度，3) データセンタ間通信の転送量の 3 つの観点から議論する．

4.2.1 ストレージ性能の均等な拡張性

分散キーバリューストアは各ノードに対して負荷が均一に分散している必要がある．なぜならば，負荷が集中しているノードがある場合，そのノードがボトルネックとなり分散キーバリューストア全体の性能を低下させるためである．また，このような状況下では，ノードを追加しても分散キーバリューストアの性能向上が小さくなるため拡張性の小さいシステムとなる．すなわち，各ノードの性能が同じ場合であれば，各ノードが担当するキー空間の大きさは確率的に均一である必要がある．

単一のデータセンタから構成される分散キーバリューストアにおいては，追加されたノードはキー空間上に確率的に均等に分散するため，各ノードが担当するキー空間の大きさも確率的に均等に分散する．したがって，ノードの追加に応じて各ノードの負荷が均等に軽減されるため，効率的な性能向上が期待できる．新たに追加されたノードにはキー空間の一部が割り当てられ，該当するキー空間を元々担当していたノードから必要なデータが移譲される．同様に，複数のデータセンタのノードを用いて分散キーバリューストアを構築する場合においても，図 4.2 に示すように各ノードの担当するキー空間の範囲は均等になっていれば効率的な性能向上が期待される．このような分散キーバリューストアにおいては，どのデータセンタに属するノードを追加しても全体の負荷分散に繋がるため，各データセンタの状況に応じて自由に資源を利用可能である．

一方，複数のデータセンタから構成される分散キーバリューストアではこのような効率的な拡張性を得られない場合がある．たとえば，データセンタごとにキー空間が分割された分散キーバリューストアでは，ノードを追加しても全体の性能向上には繋がらない．図 4.3 に，キー空間がデータセンタ毎に分割された分散キーバリューストアの例を示す．この例では，それぞれのデータセンタにキー空間が半分ずつ割り当てられ，各キー空間は更に各データセンタに属するのノードに均一に割り当てられている．このような状況においては，一方のデータセンタにいくらノードを追加したところで他方のデータセンタに属するノードの負荷は分散されないため，資源の少ないデータセンタがボトルネックとなり得る．したがって，複数のデータセンタを跨いで拡張性の高い分散キーバリューストアを構築するには，データセンタによらず各ノードを同等に扱う必要がある．

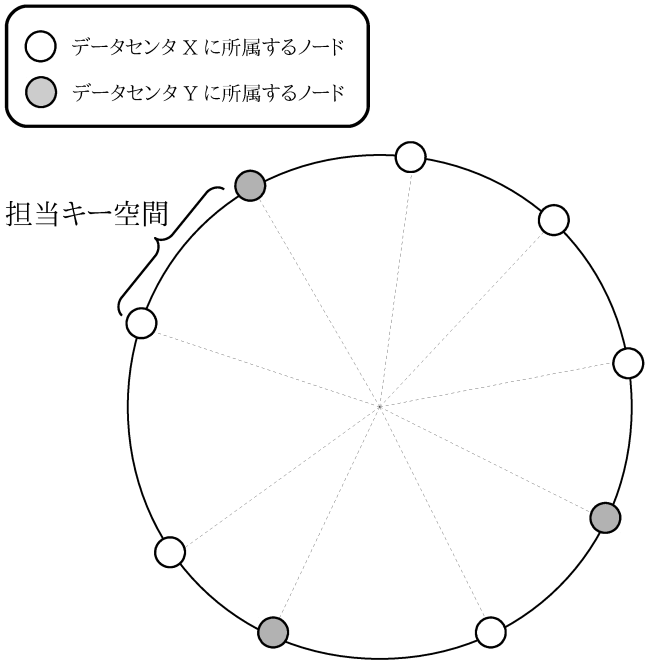


図 4.2: 一律のキー空間を持つ分散キーバリューストア

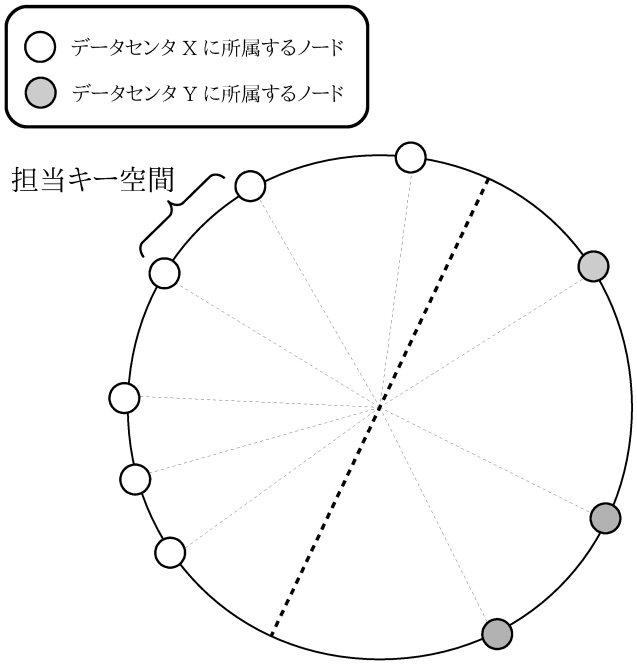


図 4.3: キー空間が分割された分散キーバリューストア

4.2.2　データセンタ間通信の頻度

複数のデータセンタ間を跨ぐ分散キーバリューストアでは，データセンタ間通信の発生を少なく抑える必要がある．なぜなら，データセンタ間通信はデータセンタ内通信に対して遅延が大きく，データ探索時間の増大に繋がるためである．データ探索時間の増大は，この分散キーバリューストアを利用するアプリケーションの応答時間の増大に繋がり，そのサービスの品質を低下させる．

図 4.6a に，実ネットワーク上におけるノードの分布を考慮しない一般的な DHT による非効率なデータ探索の例を示す．本例において，各ノードは目的のノードに近いノードを自身の保持する経路表より選択し，次のホップ先とする．探索を開始するノードと目的のノードはそれぞれ異なるデータセンタに属している．この例では，探索経路に同じデータセンタ間を跨ぐ通信を複数回含むが，一度別のデータセンタにホップした後に再び元のデータセンタに戻ることは通信遅延が大きく無駄である．このような不必要なデータセンタ間通信の発生は，DHT によるオーバレイネットワークが各ノードの属するデータセンタを考慮していないことに起因する．これはデータ探索に要するを不必要に増大させるため，複数のデータセンタ間を跨ぐ分散キーバリューストアには不要なデータセンタ間通信を避けるための仕組みが必要である．

4.2.3　データセンタ間通信の転送量

複数のデータセンタから構成される分散キーバリューストアにはデータセンタ間通信路を介したデータ転送量を減少させる仕組みが必要である．なぜならば，一般的に，データセンタ間通信路はデータセンタ内通信路と比較して狭帯域であり，同じ量のデータの転送に要する時間がより大きく，スループットを低下させるためである．

データ読み込み時におけるデータ転送量を減らすために効果的なアプローチのひとつとして，別のデータセンタに存在するデータをローカルに複製して保持する方法がある．これにより，そのデータを必要とするノードは，他のデータセンタからではなく同じデータセンタ内に保持されているレプリカを取得し，狭帯域なデータセンタ間通信の利用を避ける事ができる．しかし，このアプローチは全てのデータオブジェクトを各データセンタに複製するため大量のストレージ資源を消費する．n 箇所のデータセンタにて分散キーバリューストアが稼働する場合，

全体では本来保存されるデータサイズの n 倍のストレージを消費する．

また，こうしたデータ複製を行った場合，データ更新時に各複製を更新するためにデータセンタ間の転送が必要になるという問題がある．したがって，更新頻度の高いワークロード下においては，更新に伴うデータ転送量を抑えるための対策が必要である．たとえば，配置するレプリカの数を減らすことや，更新データの一部のみを転送することなどによって，転送量を軽減することができる．

ストレージ使用量とデータ転送量のバランスを取るために，分散キーバリューストアにはより少ないストレージ使用量でデータ転送量も軽減する仕組みが必要である．

本研究では，一般的な分散キーバリューストアの操作におけるデータセンタ間通信を減少させることに着目している．すなわち，データセンタやノードに発生した障害やその復旧に伴い発生するデータセンタ間通信を減少させることを目的とはしていない．このようなデータセンタ間通信を減少させることも興味深い事案であるが，本論文では対象としない．

4.3 設計

本節では，複数のデータセンタ間を跨ぎ単一のキー空間を持つ分散キーバリューストア構築手法 Pangaea の設計について説明する．まず，提案手法の概要について述べ，提案手法の位置付けと効果を明確にする．つづいて，データセンタ間通信に伴う問題を解決するための要素技術，1) Multi-Layered Distributed Hash Table (ML-DHT) および 2) Local-first Data Rebuilding (LDR) の設計について説明する．

4.3.1 概要

Pangaea は複数のデータセンタに属するノードを統合して単一の分散キーバリューストアを構築する手法である．本手法は，単一のデータセンタの制約を受けることなく，また，サービスを停止すること無く動的にノードの追加および離脱が可能であり，高い拡張性と伸縮性を実現する．また，ノードの追加に伴うデータの転送は，各ノードが担当する領域に限定されるため，すべてのデータを転送する必要のある完全複製方式に比べ敏捷性が高い．これはキー空間が Consistent Hashing [132] を用いて構築されているためである．Consistent Hashing を用いると，新たにノードが加入や離脱をする度に，全てのノードの担当キー空間の再割当てを伴わずに

済む．さらに，DHT により一意に定まる位置にデータを格納するため，複製との同期などは不要であり一貫性は高い．

Pangaea の要素技術である ML-DHT と LDR は高遅延かつ狭帯域であるデータセンタ間通信が分散キーバリューストアの性能に及ぼす悪影響を低減する．図 4.1 に示したように，ML-DHT および LDR はそれぞれルーティング層およびデータフェッチ層で動作する．本手法を適用した分散キーバリューストアを利用するアプリケーションは，それが単一のデータセンタのノードのみで構成されているのか複数のデータセンタのノードから構成されているのかを意識する必要は無い．なぜなら，アプリケーションが利用するデータの保存場所は抽象化されており，また，性能低下に繋がるデータセンタ間通信を少なく抑える仕組みを備えているためである．ML-DHT はデータ探索時の応答遅延の増大を抑制する仕組みを備える．これはデータ探索時に冗長なデータセンタ間通信が発生することを許さない．LDR は狭帯域であるデータセンタ間通信路を用いたデータ転送量を低減させる仕組みを備える．これはキーバリューストアに対する読み書きに伴うデータセンタ間通信を抑制する．第 4.3.2 項および第 4.3.3 項にて，ML-DHT と LDR についてそれぞれ説明する．

また，ML-DHT はストレージの均一な拡張性の実現にも貢献する．分割されたキー空間と異なり，ML-DHT は一律のキー空間を提供する（図 4.2）．すなわち，キー空間はデータセンタ毎に分割されることなく，全てのノードが同様の扱いを受ける．ML-DHT における新たなノードの追加は，分散キーバリューストア全体の性能向上に繋がる．例えば，図 4.2 においてデータセンタ X にノードが追加された場合，そのノードは一部のキー空間の割り当てを受け，隣接ノードから対象のデータを受け取る．もし隣接ノードがデータセンタ X 以外のデータセンタ Y であった場合，データセンタ X に追加されたこのノードによってデータセンタ Y のノードが負荷分散の恩恵を受けることとなる．この際，担当するキー空間の移譲のためにデータセンタ間通信が発生するが，これは頻繁に起こらないと想定する．なぜなら，データセンタで運用される分散キーバリューストアにおいては，一般ユーザの PC 等から構成される Peer-to-Peer 型システムと異なり，ノードの加入 / 離脱の障害やメンテナンスに起因するため頻度が低いと想定されるためである．

4.3.2 Multi-Layered Distributed Hash Table

Multi-Layered Distributed Hash Table (ML-DHT) は，一定の基準でまとめられた

表 4.1: 記号の定義

記号	定義
L	ML-DHT が構成するオーバレイネットワークレイヤの数．$L \in \mathbb{N}$（$\mathbb{N}$ は自然数 ）
l	ML-DHT が構成するオーバレイネットワークレイヤのひとつ．$l \in \mathbb{N}, 0 \leq l < L$
G	全ノードの集合
$G_{n,l}$	第 l 層にてノード n が属する集合．
$n.tables[l]$	ノード n が保持する，第 l 層用の経路表．$G_{n,l}$ に含まれるノードを管理対象とする．
$n.intervals[l]$	第 l 層における ノード n の担当キー空間
$n.responsible$	ノード n が実際にデータの保持を担当するキー空間．$n.intervals[0]$ と一致する．

ノードの集合に対し，同じ集合内のノードを優先的に次の経路として選択可能にする DHT 拡張手法である．ML-DHT において各ノードは，オリジナルの DHT の経路表に加え，同じ集合のノードのみを含む経路表を追加で保持する．経路探索は可能な限り同じ集合内で実行され，同じ集合内においてそれ以上目的のノードに近づくことができない場合のみ別の集合に属するノードを次の経路として選択する．すなわち，ML-DHT を用いると，2 つの集合の間を往復するような経路を取ることはない．たとえば，同じデータセンタに属するノードを集合とし ML-DHT を用いることで，データセンタ間を不必要に往復する探索の発生を防ぐことができる．したがって，ML-DHT を用いることにより，第 4.2.2 項で述べたような非効率的なデータ探索を避けることができる．また，本手法はマルチホップ探索を行う DHT であれば一般的に適用可能であり，任意のノードへの到達性とデータの可用性をオリジナルの DHT と同等の水準で保証する．

自然数であることを明記した．

本手法の説明に用いる各記号を表 表 4.1 に，集合 $G_{n,l}$ の関係について具体例を図 4.4 に示す．本手法において各ノードは，実ネットワーク上における距離が一定の基準より小さいノードの集合を管理対象とする L 個の経路表を保持し，L 層のオーバレイネットワークレイヤを構成する．本論文では，第 0 層を最下位レイヤと定義し，全体集合を管理対象とする．一方，l が大きいレイヤを上位レイヤと定義し，より小さな集合を管理対象とする．すなわち，上位層における集合は，実

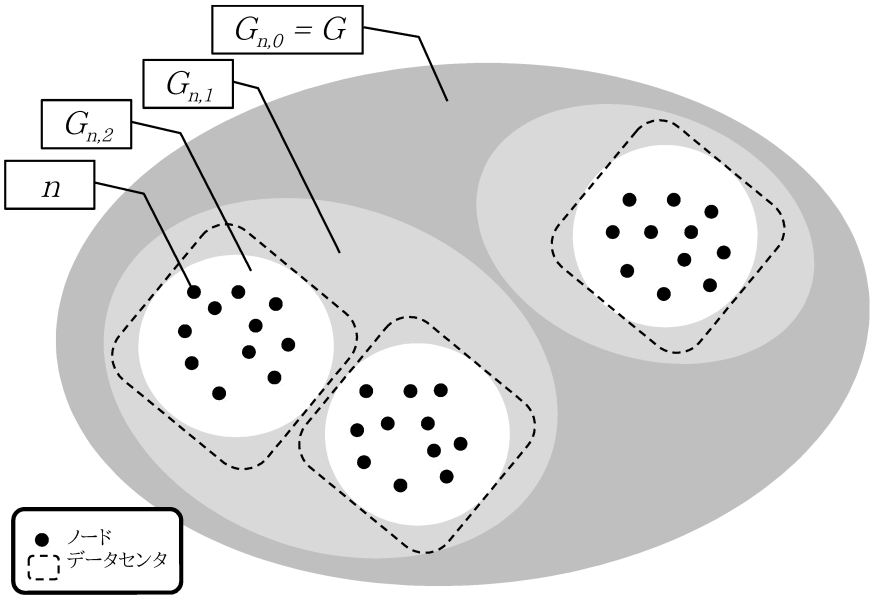


図 4.4: ノード分布と ML-DHT におけるレイヤ構成の例 ($L = 3$)

ネットワーク上でより近いノードから構成される．また，集合 $G_{n,l}$ は一般的に次の各性質を満たす．但し，$n_1 \neq n_2$ とする．

$$G_{n,k+1} \subset G_{n,k} \quad \text{for} \quad \forall k \in \mathbb{N}, 0 \leq k < L-1 \tag{4.1}$$

$$G_{n,0} = G \quad \text{for} \quad \forall n \in G \tag{4.2}$$

$$G_{n_1,l} = G_{n_2,l} \quad \text{for} \quad n_1 \in G_{n_1,l} \wedge n_2 \in G_{n_1,l} \tag{4.3}$$

$$G_{n_1,l} \cap G_{n_2,l} = \varnothing \quad \text{for} \quad n_1 \in G_{n_1,l} \wedge n_2 \notin G_{n_1,l} \tag{4.4}$$

ML-DHT を構成する各ノードは各レイヤについて担当のキー空間 $n.intervals[l]$ を割り当てられるが，これはオーバレイネットワークを管理するためのものであり，実際に管理を担当するデータは $n.responsible(= n.intervals[0])$ に含まれるものが対象となる．

また，各経路表の更新処理は，対象となるノードの集合が異なる点を除き，それぞれオリジナルの DHT と同様のアルゴリズムで行う．すなわち，全てのノードを対象としている経路表 $n.table[0]$ はオリジナルの DHT と完全に同等であり，これにより全ノードへの到達性と可用性を同等の水準で保証する．

ML-DHT へのノードの加入およびデータの探索は以下に示す手順で行う．

加入処理 (Join)

ML-DHT において，新たに加入するノード n_{new} は L 個の経路表全てについて加入処理を行う．加入処理に当たり，n_{new} は $G_{n_{\mathrm{new}},L-1}$ から任意のノードをブートストラップノード n_{bs} として選出する．加入処理に伴う経路表 $n.tables[l]$ の更新手順はオリジナルの DHT と同様であるが，各層の独立性を保証するため，管理対象層の異なる経路表を混同しないよう行う．経路表 $n.tables[l]$ は次の手順で更新される．

1. n_{new} が n_{bs} に対し $n_{\mathrm{new}}.tables[l]$ について加入要求を送信する．
2. n_{bs} は $n_{\mathrm{bs}}.tables[l]$ を用いて，$n_{\mathrm{new}}.intervals[l]$ を現在担当しているノード n_{cur} を特定する．
3. n_{bs} は n_{new} に n_{cur} の情報を送信する．
4. n_{cur} は n_{new} に対し，オリジナルの DHT と同様の方法で $n_{\mathrm{cur}}.intervals[l]$ から $n_{\mathrm{new}}.intervals[l]$ を移譲，加入処理を完了する．但し，移譲するキー空間に該当する保存データの移送は $l = 0$ についてのみ行う．

もし第 l 層に n_{bs} となり得るノードが存在しない場合は，n_{new} を最初のノードとして $n_{\mathrm{new}}.tables[l]$ の構築を行う．

探索処理 (Look-up)

ML-DHT は，経路表 $n.tables[l]$ を用いて到達可能なノードが集合 $G_{n,l}$ のノードであることを利用し，同じ集合間を往復せずに目的のキーを探索する．

図 4.5 に ML-DHT におけるキー id 探索処理の擬似コードを示す．各ノードはキー id を探索する際に，より上位の（l が大きい）層の経路表を優先的に利用し，その層で担当のノードが見つからなかった場合に下位の経路表を利用する．第 l 層における探索処理はオリジナルの DHT と同様に行い $id \in n.intervals[l]$ となるノード n が見つかるまで実行するが，この間経路に $G_{n,l}$ 以外のノードを含むことはない．$id \in n.intervals[l]$ を発見した場合，$id \in n.responsible$ であるかを確認する．この確認処理は，ノード n が自身のローカル情報を確認するのみで通信は伴わない．ここで $id \in n.responsible$ の場合，ノード n が担当のノードであるため探索を終了する．一方 $id \notin n.responsible$ の場合，ノード n は担当ノードではないが

```
// node n has routing tables for each layer (L is # of layers)
// each table is dealt with following an original DHT algorithm
n.tables = {table_0, table_1, … , table_L-1};

// ask node n to find responsible node for specified id
n.find_responsible_node(id)
    while (id not in n.intervals[0])
        for layer = L-1 downto 0
            n' = get_next_candidate(id, layer);
            if (n' exists)
                return n'.find_responsible_node(id);
    return n; // n is a responsible node for the id

// get next node from layer^th routing table of local node
n.get_next_candidate(id, layer)
    // choose next node n' from n.tables[layer]
    // following the original DHT algorithm
    n' = tables[layer].get_next_candidate_from_table(id);
    if (n' exists and n' != n)
        return n'; // node n' is more suitable than node n
    return; // node n is responsible for the id in this layer
```

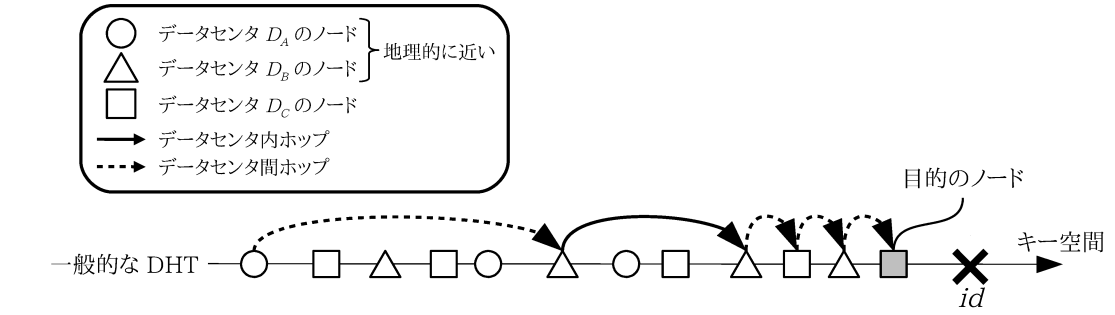
図 4.5: ML-DHT におけるキー id 探索処理の擬似コード

第 l 層にはそれ以上探索対象となるノードが残っていないため，下位層にて次のホップ先を探す．下位層にてホップした後は，同様に探索を行う．

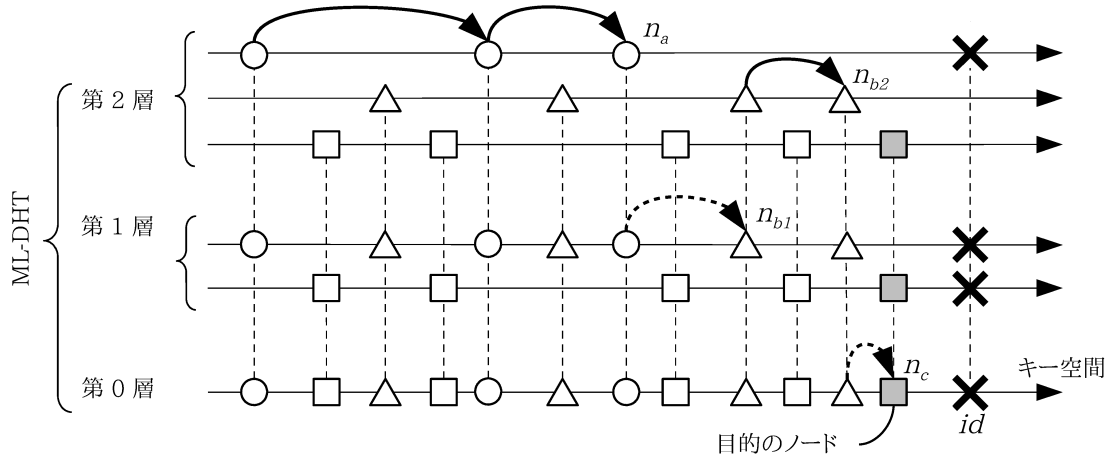
また，ML-DHT における探索の経路長はオリジナルの DHT と同じかそれより短い．これは，$G_{n,l} \subset G$ より $|G_{n,l}| < |G|$ であり，上位層では 1 ホップで目的のノードにより近づけるためである．一方で，ML-DHT では各層に対して経路表を保持する必要があるため，オリジナルの DHT に比べ L 倍の領域を経路表のために確保する必要がある．

ML-DHT は反復型 (Iterative) および再帰型 (Recursive) の探索いずれにも適用可能であり経路長を短縮できるが，本論文ではより利点の多い再帰型探索を用いる．反復型の場合，探索が一度でも別の集合にホップするとそれ以降は集合内でのホップであっても常に集合を跨いだ通信を伴う．一方，再帰型の場合，常にホップ元とホップ先のノード間で通信するため，集合内でのホップは集合を跨いだ通信を伴わない．すなわち，再帰型の探索の方がより ML-DHT によるデータセンタ間通信頻度の削減効果が得られる．

図 4.6b に ML-DHT によりデータセンタ間通信の頻度を削減する例を示す．本例において，各ノードは目的のノードに近いノードを自身の保持する経路表より



(a) 一般的な DHT における探索処理例



(b) ML-DHT に拡張した場合における探索処理例 (L = 3)

図 4.6: データセンタ間を跨ぐ環境における探索処理の比較

選択し，次のホップ先とする．また，各層 0, 1, 2 はそれぞれ全てのノード，近隣地域のノード，データセンタ内のノードを管理対象とする．探索処理は次のように進行する．

1. $id \in n_a.intervals[2]$ となるノード n_a まで第 2 層（D_A 内）を探索する．
2. $id \notin n_a.responsible$ であるため，$n_a.tables[1]$ を用いて n_{b1} へホップする．
3. $id \in n_{b2}.intervals[2]$ となるノード n_{b2} まで第 2 層（D_B 内）を探索する．
4. $id \notin n_{b2}.responsible$ であるが，$n_{b2}.tables[1]$ からは次のホップ先を得られないため，$n_{b2}.tables[0]$ を用いて n_c へホップする．
5. $id \in n_c.responsible$ であるため，探索を終了する．

このように ML-DHT では集合間を往復するホップは発生しないため，データセンタの拠点数以上のデータセンタ間ホップが発生することはない．また，探索処理は一般的な DHT よりも少ない数のデータセンタ間ホップで完了することが可能である．データセンタの拠点数が増えると，データセンタ間ホップの最大値も増加するため，下位層で複数のデータセンタを含む集合を設ける等の対応を取ることで，データセンタ間ホップが生じても比較的近距離のデータセンタが選択されやすくなる．

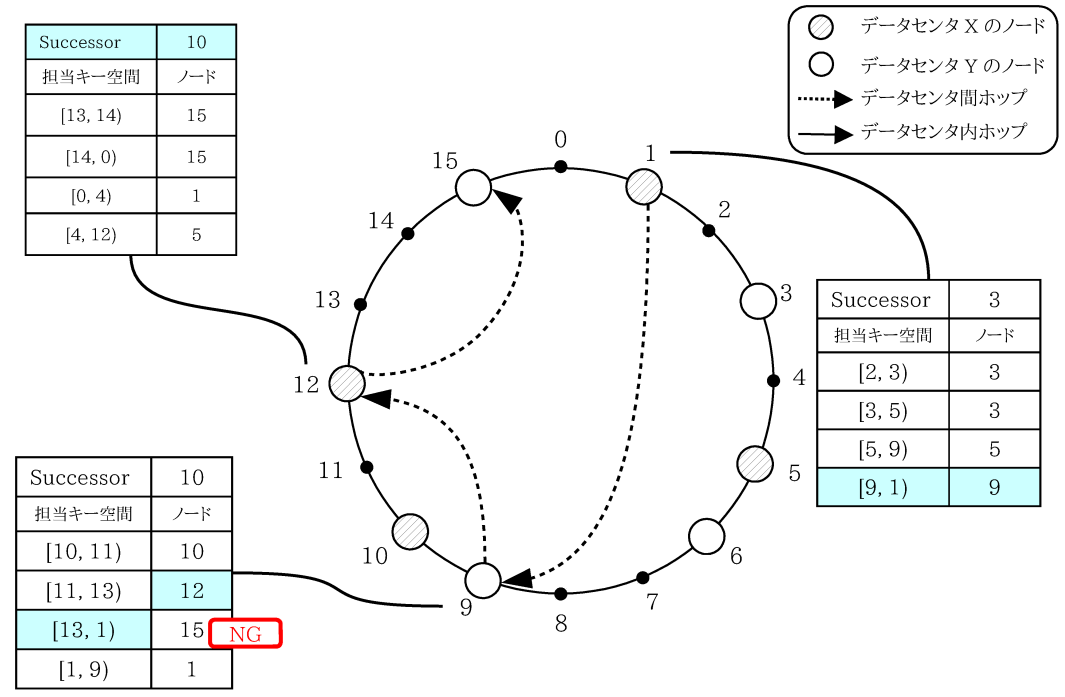
ML-Chord

ここでは，DHT を ML-DHT に拡張する例として，代表的な DHT アルゴリズムである Chord [114,115] を多層化した ML-Chord について述べる．ここでは各ノードが持つ経路表の数 L を 2 とし，各経路表をグローバル経路表 ($l = 0$) とローカル経路表 ($l = 1$) と呼ぶ．また，第 1 層におけるノードの集合はデータセンタ単位とする．

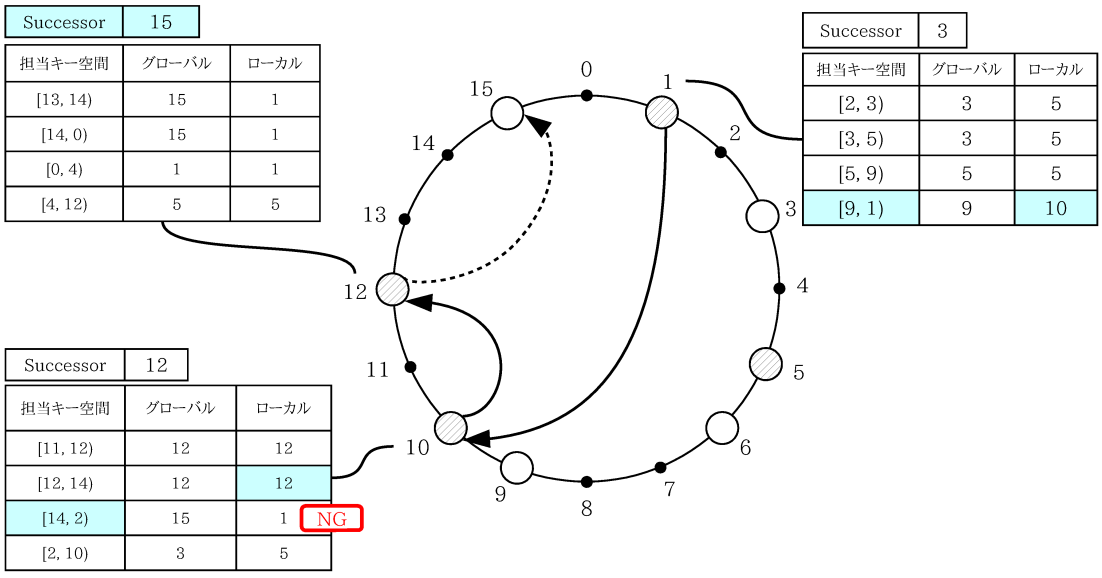
ML-Chord に加入するには，n_{new} は n_{bs} に加入要求を送信する．n_{new} および n_{bs} は，通常の Chord と同様のアルゴリズムでグローバル経路表をそれぞれ初期化および更新する．双方が同じデータセンタに属する場合，ローカル経路表に含まれる情報を用いて，それぞれのローカル経路表を初期化および更新する．一方，双方がそれぞれ異なるデータセンタに属する場合，n_{new} はローカル経路表を自身のみ存在するものとして初期化し，n_{bs} はローカル経路表に対して何も処理を行わない．

図 4.8 に ML-Chord における探索処理の擬似コードを示す．各ノードはクエリの転送先，すなわち次のホップ先を選択する際にローカル経路表を優先的に利用する．各ノードは，次のホップ先として適切なノードをローカル経路表から発見できない場合のみ，グローバル経路表を用いる．

図 4.7 は Chord 型の DHT と ML-Chord におけるルーティングの例の比較である．各例とも，ノードの構成は同様で，ノード 1, 5, 10 および 12 はデータセンタ X，残りはデータセンタ Y で稼働している．また，各例とも，データセンタ Y 所属ノード 15 の管理担当データ 14 をデータセンタ X 所属ノード 1 が取得する場合を図示している．すなわち，最低でも 1 度はデータセンタ間を跨ぐホップが必要である．Chord 型の DHT では，3 つのホップ全てがデータセンタ間を跨いでおり，探索に伴う応答遅延が大きくなっている．このような無駄なホップが発生しているのは，経路表が各ノードの属するデータセンタ（ローカルまたはリモート）



(a) Chord 型 DHT における探索処理例



(b) ML-Chord における探索処理例

図 4.7: Chord 型 DHT と ML-Chord における探索の比較

を考慮せずに次のノードを示すためである．一方，ML-DHT では 3 つのホップのうち最後の 1 つのみがデータセンタ間を跨いでいる．これは，ローカル経路表がデータセンタ間ホップの実行を可能な限り先送りする役割を果たしているためで

```
#define GLOBAL_LAYER 0
#define LOCAL_LAYER 1

// node n has finger table for each layer
n.finger_tables = {global_finger_table, local_finger_table};

// ask node n to find id's successor
n.find_successor(id)
    n' = find_predecessor(id);
    return n'.successor;

// ask node n to find id's predecessor
n.find_predecessor(id)
    n' = n;
    while (id not in (n', n'.successor])
        n' = n'.closest_preceding_finger(id);
    return n';

// return closest finger preceding id
n.closest_preceding_finger(id)
    for l = LOCAL_LAYER downto GLOBAL_LAYER
        for i = m - 1 downto 0 // m is # of finger table's entries
            if (finger_tables[l][i].node in (n, id))
                return finger_tables[l][i].node;
    return n;
```

図 4.8: ML-Chord におけるキー id 探索処理の擬似コード

ある．図 4.7b において各ノードは以下の手順で次のノードを選定している．

1. 次のノード候補として，ノード 1 のローカル経路表はノード 10 を，グローバル経路表はノード 9 を示している．ノード 10 の方がより目的のノードに近いため，ノード 1 はノード 10 を選択する．

2. 次のノード候補として，ノード 10 のローカル経路表はノード 1 を，グローバル経路表はノード 15 を示している．ノード 10 は目的のデータ 14 とノード 1 の間にノード 15 が存在することを知っているため，ノード 1 を選択すると目的のキーを通過してしまうこともわかる．したがって，ノード 10 はローカル経路表においてキーに近い候補としてノード 12 を選択する．

3. 次のノード候補として，ノード 12 のローカル経路表はノード 1 を示しているが，ノード 15 がノード 1 の手前にあることを知っている．直前の手順と同様に，ノード 12 はローカル経路表から次の候補の選定を試みるが，適切

なノードが含まれないため選択できない．このような場合に初めて，ノード 12 はグローバル経路表における候補であるノード 15 を次のノードとして選択する．本例における，データセンタ間ホップはこの一度のみである．

4.3.3 Local-first Data Rebuilding

Local-first Data Rebuilding (LDR) はデータセンタ間のデータ転送量を小さく抑えつつ，目的のデータを取得するための手法である．LDR は Erasure Coding を用いることで，データセンタ間のデータ転送量とストレージの使用量を任意に調整可能にする．本手法は，Weatherspoon と Kubiatowicz [133] の研究に影響を受けている．彼らは Erasure Coding とレプリケーションについて，耐障害性とストレージ使用量の観点で評価を行った．

LDR は 3 つのステップから成る．まず Erasure-coding Step では，Erasure Coding を用い保存対象のデータを複数のチャンクと呼ばれるデータ断片に分割する．これは分散キーバリューストアにデータを `put` する際に，`put` を行うノードが実行する処理である．次に Uniform Data Putting Step では，キー空間へ均一に分散するようにして各チャンクを保存する．本研究では，この均一性は既存のハッシュアルゴリズムによって実現されるものとする．最後に Local-first Data Fetching Step では，ローカルのデータセンタに属するノードが保持するチャンクを優先的に利用して元のデータの復元を行う．Erasure-coding Step と Uniform Data Putting Step はデータを保存 (`put`) する際に実行され， Local-first Data Fetching Step はデータを取得 (`get`) する際に実行される．

Erasure-coding Step において，保存対象のデータは冗長性を持った m 個のチャンクに分割される．Erasure Coding を用いることで，元のデータは m 個のチャンクのうち，どの k 個のチャンクを用いても復元が可能となっている．($k \leq m$)

Uniform Data Putting Step において，m 個のチャンクはそれぞれ分散キーバリューストアに保存される．データセンタ間通信の機会を削減させるため，各チャンクはキー空間へ均一に分散して保存されなければならない．これを実現するために，LDR では各チャンクはハッシュ関数を用いて生成した値をキーとすることで確率的に均一に分散して保存を行う．各チャンクの保存処理自体は通常のキーバリューストアと同様の手順で行う．

Local-first Data Fetching Step では，データを要求するノードは k 個のチャンクを収集する．この際，任意の k 個のチャンクから元のデータを復元できることと，

各ノードがML-DHTのローカル経路表を用いて同じデータセンタに属するノードを優先的に探索できることが重要となる．データを要求するノードはまず，k 個のチャンクの収集を，ローカル経路表のみを用いて取得することを試みる．k 個以上のチャンクを収集できた場合は，それらを用いて元のデータの復元処理を行い処理を終了する．この時データセンタ間を跨ぐ通信は一切発生しないため，処理は短時間で終了する．一方，k 個未満のチャンクしか収集できなかった場合は，グローバル経路表も用いてチャンクを収集する．この際リモートのデータセンタからの取得が必要なチャンクの数は最大で k 個であり，1 個以上ローカルのデータセンタから取得できていた場合データセンタ間のデータ転送量はその分削減される．

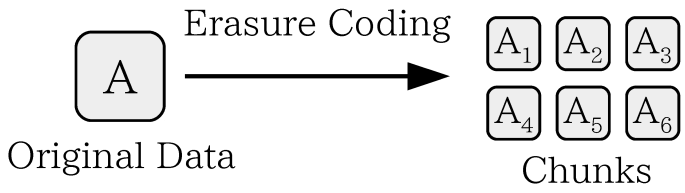
データセンタ間を跨いで転送されるチャンクの数は，ローカルのデータセンタに属するノードの比率に比例して減少する．これは各チャンクのキーがハッシュアルゴリズムによってほぼ均一に分散されているためである．

図 4.9 および 図 4.10 に LDR がデータセンタ間のデータ転送量を削減する例を示す．本例において，Erasure Coding のパラメータ (m, k) は $(6, 4)$ とし，また，データセンタ X および Y に属するノード数の比率は 2:1 とする．この場合，各データは次のように処理される．

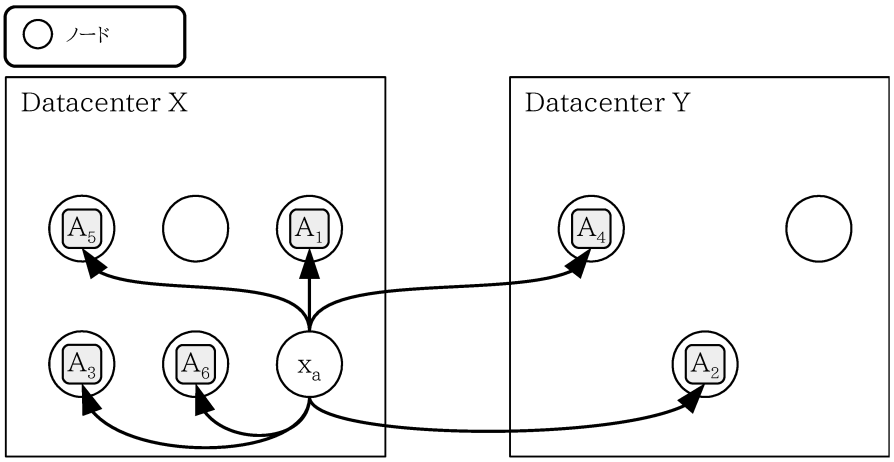
1. 元のデータを Erasure Coding を用いて 6 個のチャンクに分割し，生成したチャンクを担当ノードに配置するこの際，チャンクは確率的に均一にキー空間上に配置されるため，2:1 の比率でデータセンタ X, Y に配置される．すなわち，6 個のチャンクのうち 4 個はデータセンタ X へ，残りの 2 個はデータセンタ Y に配置される可能性が高い (図 4.9).

2. データセンタ X に属するノードは元のデータ復元に必要な 4 個のチャンク全てをローカルのデータセンタ内で収集することができる．一方，データセンタ Y に属するノードはローカルのデータセンタ内の収集では 2 個のチャンクが不足するため，これらをリモートのデータセンタ X から取得する必要がある．しかし，2 個のチャンクのみをリモートのデータセンタ X から取得すれば十分であり，元のデータ全てをリモートから取得する場合よりも少ない転送量で目的を達成できる (図 4.10)

もし単純分割（ストライピング）を用いた場合，データセンタ間のデータ転送量は LDR よりも大きくなる．これは単純分割で生成されたチャンクから元のデータを復元する場合は 6 個すべてが必要となるためである．また，データセンタの



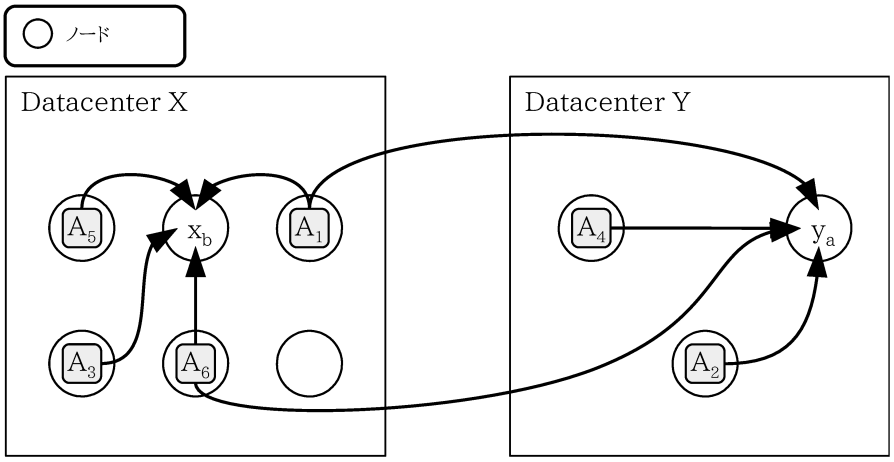
(a) Erasure Coding によるチャンクの生成



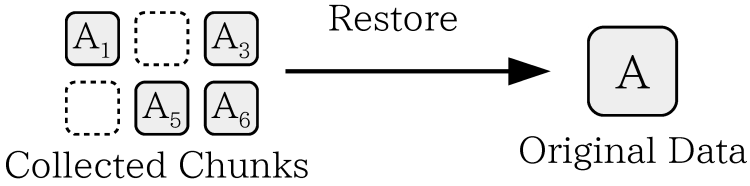
(b) Uniform Data Putting

図 4.9: LDR によるデータの加工と配置の例 $(m, k) = (6, 4)$

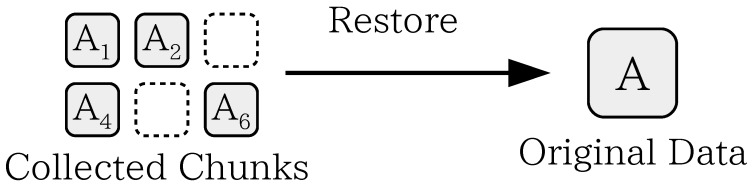
拠点数が増加すると，他のデータセンタに配置されるチャンクの数が増加するため，データセンタ間のデータ転送量が増加する．LDR によるデータセンタ間の転送量の削減効果については 第 4.5 節にて評価する．



(a) Local-first Data Fetching



(b) ノード x_b におけるデータの復元



(c) ノード y_a におけるデータの復元

図 4.10: LDR によるデータ取得の例 $(m, k) = (6, 4)$

4.4 実装

オーバレイ構築ツールキット Overlay Weaver 0.10.3 [111–113] を用いて提案手法のプロトタイプを Java および Python により実装した．ML-DHT の実装例として，ML-Chord を Overlay Weaver 上に実装した．ML-Chord は Overlay Weaver に含まれる Chord の実装を拡張することで実装した．また，LDR の実装には Erasure Coding の実装である Zfec-1.4.24 [134] を用いた．LDR におけるエンコード関数は m, k 2 つのパラメータを引数とし，m は元のデータから生成するチャンクの数，k は元のデータを復元するために必要なチャンクの数とした．各チャンクのキーには元のデータを `put` する際に用いられたキーに添字として通し番号を付加したも

のを用いた．たとえば，元のリクエスト `put("foo", value)` はチャンクを生成した後に，`put("foo_1", chunk_1)`, `put("foo_2", chunk_2)`, ..., `put("foo_m", chunk_m)` という形で処理される．

4.5 評価

本節では，提案手法の要素技術である ML-DHT および LDR がデータセンタ間通信による分散キーバリューストアの性能低下を軽減することを示すための評価実験について述べる．提案手法の有用性を確認するために，シミュレーション環境と実際のインターネット環境の 2 つを用い，データ探索の所要時間とデータ転送量について評価した．また，これら 2 つの評価の前に，用いたインターネット環境がどのようなものであったか，通信遅延とスループットについても計測している．

シミュレーション環境には，500 台のノードが稼働するデータセンタを 2 つ用意した．シミュレーションにはシングルコア 3.00 GHz Intel Xeon CPU および 2 GB の主記憶を備えた計算機を用い，Linux-3.2 および Java-1.6 を使用した．

実際のインターネット環境には，Amazon Elastic Computing Cloud (EC2) [83] と Virtual Private Cloud (VPC) [135] を用いた．東京リージョンおよびサンパウロリージョンに VPC を用意し，OpenVPN-2.3.2 [136] を用いてデータセンタ間通信を暗号化した．東京リージョンには 2 つのマイクロインスタンス (t1.micro) を用意し，サンパウロリージョンには 1 つのマイクロインスタンス (t1.micro) を用意した．

本節で示す実験には，データセットとして一様分布となるようランダム生成した 10,000 個のキーとデータのペアを用いた．各キーは 160 ビットのハッシュ値で，データにはランダム生成した 10 KB のバイト列を用いた．このデータセットの生成のために，キーバリューストアへ `put` および `get` 要求を発行するワークロード生成器を実装した．

また，個別に指定する場合を除き，ML-Chord のレイヤ数は $L = 2$，LDR のパラメータ設定は $(m, k) = (10, 5)$ とした．

4.5.1 データセンタ内外の通信遅延とスループット

Pangaea の評価に先立ち，通信遅延とスループットの参考とするため，ラウンドトリップ時間 (RTT) とネットワーク帯域幅を，東京リージョン内およびリージョ

ン間にて PING メッセージと iperf-2.0.5 [137] を用いて計測した．iperf は TCP および UDP での帯域幅を計測するためのツールである．各計測は 2013 年 10 月 10 日に行った．東京リージョン内および東京–サンパウロ間の RTT はそれぞれ 0.391 ms と 384 ms であった．すなわち，この環境におけるデータセンタ間通信はデータセンタ内通信よりも通信遅延が約 1,000 倍大きかった．また，東京リージョン内および東京–サンパウロ間のネットワーク帯域幅はそれぞれ 147 Mbps と 3.29 Mbps であった．すなわち，この環境におけるデータセンタ間通信はデータセンタ内通信の約 2.2 % の帯域幅であることがわかった．

4.5.2 データ探索の所要時間

ML-DHT がデータ探索に要する時間を削減することを示すために，実際のインターネット環境にて Chord と ML-Chord それぞれの探索時間を比較した．ここでは，あるデータの担当ノードの探索処理を開始してから目的ノードを発見するまでに要した時間を探索時間と定義する．ML-Chord は ML-DHT の実装例である．図 4.11 に探索時間の平均値の比較を示す．エラーバーは最大値および最小値を示す．ML-Chord の探索時間は Chord より約 74 % 小さかった．これは ML-DHT のグローバル経路表とローカル経路表を用いることで，不必要なデータセンタ間ホップが発生しないようなデータ探索を行っていたためである．

ML-Chord が実際にデータセンタ間ホップの回数を低減していたことを示すために，シミュレーション環境にて同じワークロードを実行し Chord と ML-Chord の探索経路を解析した．図 4.12 にデータセンタ間ホップ数および総ホップ数の平均値を示す．エラーバーは最大値および最小値を示す．ML-Chord は Chord より約 74 % 少ないデータセンタ間ホップ数で目的のノードに到達していたことがわかった．そして，ML-Chord は Chord より約 30 % 少ない総ホップ数で目的のノードに到達していたこともわかった．総ホップ数も減少したのは，ローカル経路表を用いることで，スキップリスト [138] のようにグローバル経路表にのみ含まれるノードを飛ばして目的のキーへ近づくことが可能なためである．これらの結果から，複数のデータセンタを跨ぐ分散キーバリューストアにおける探索時間は，データセンタ間の通信遅延の影響をより大きく受けることがわかる．また，ML-Chord は Chord と異なり，データセンタ間ホップが最大 1 回であったことから，データセンタ間を無駄に往復するホップを一切行っていないことがわかる．

さらに，本実験の条件設定と異なる場合における提案手法の効果を評価するため，

ML-Chord と Chord におけるデータセンタ間ホップ数の期待値 $E_{\mathrm{ML-Chord}}, E_{\mathrm{Chord}}$ を算出した (式 4.5, 4.6)．ここで，d はデータセンタの数，h は一回の探索処理に含まれるデータセンタ間ホップ数，r は探索経路長と定義し，簡単のために，各データセンタに属するノードの数は同じ，キー空間上におけるノードの分布は均一であるとする．$E_{\mathrm{ML-Chord}}$ は d にのみ応じて増加する．すなわち，ノード数が増加してもデータセンタの数が増加しなければデータセンタ間ホップの機会が増加することはない．一方，E_{Chord} は d および r に応じて増加する．Chord において r はノード数 N に対し $O(\log N)$ で増加するため，E_{Chord} はノード数とデータセンタ数に応じて増加すると言い換えることができる．したがって，データセンタ数に対しノード数が増加するほど，ML-Chord の Chord に対するデータセンタ間ホップ数削減効果が大きくなる．本実験の条件は $d = 2, N = 1000$ であったが，より多くのノードを利用する環境においては d に対し N がより大きく異なることが想定され，その場合は本手法による性能向上を見込むことができる．

$$E_{\mathrm{ML-Chord}} = \frac{\log d}{2} \tag{4.5}$$

$$E_{\mathrm{Chord}} = \sum_{h=1}^{r} {}_rC_h \left(\frac{1}{d}\right)^{r-h} \left(1 - \frac{1}{d}\right)^{h} h \tag{4.6}$$

4.5.3 データ転送量

LDR によって，分散キーバリューストアの管理者がストレージ使用量とデータセンタ間のデータ転送量を調整可能となることを示すために 2 つの評価実験を行った．これらの実験におけるパラメータ設定 (m, k) は 第 4.3.3 節および第 4.4 節における定義と同様である．第一の実験では，ストレージ使用量とデータセンタ間の転送量がパラメータ設定によってどのように変化するかを比較した．図 4.13a, 図 4.13b はそれぞれストレージ使用量とデータ転送量を示す．各図における各格子点は，パラメータ設定 (m, k) に対応する．各図より，LDR のパラメータ設定を変えることで，ストレージ使用量とデータセンタ間の転送量を細粒度に調整できることを確認した．m または k を増加させると，データ転送量は減少するがストレージ使用量は増加した．反対に，m または k を減少させると，ストレージ使用量は減少したがデータ転送量は増加した．

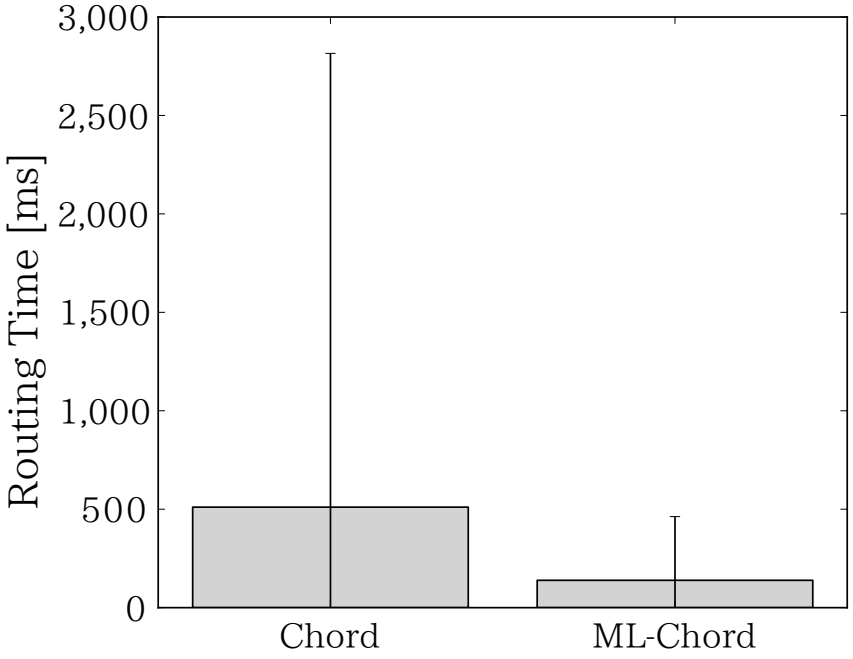


図 4.11: 実際のインターネット環境における探索時間

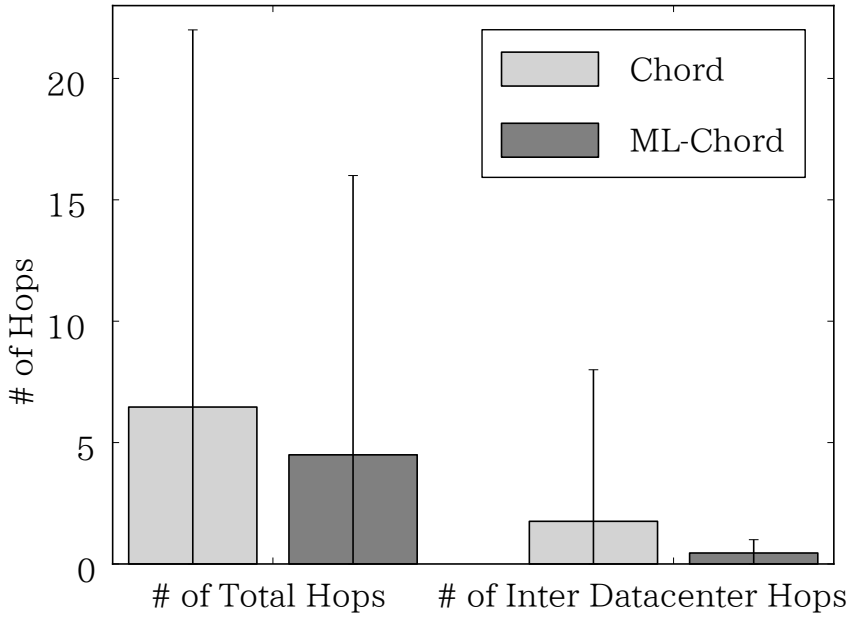
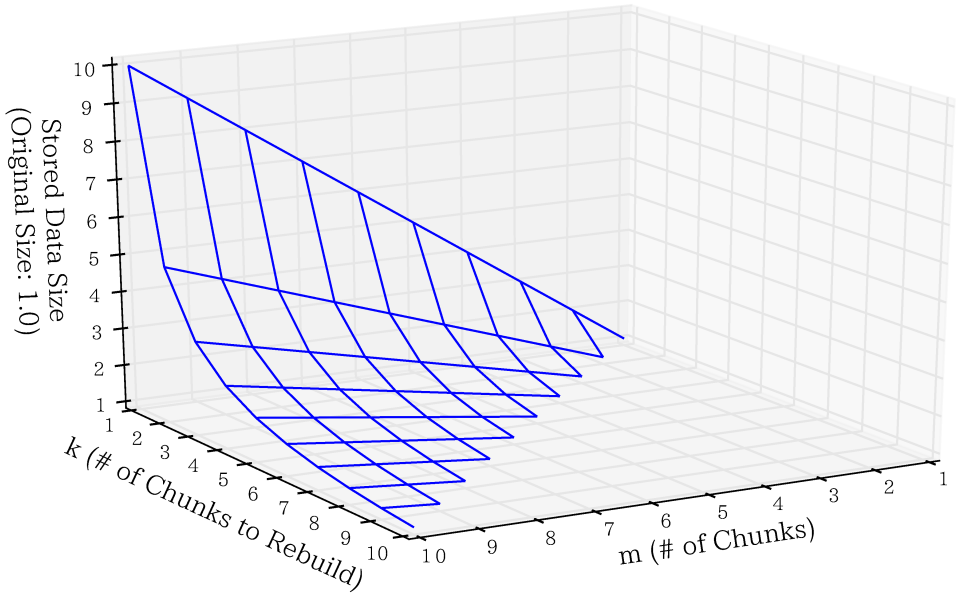


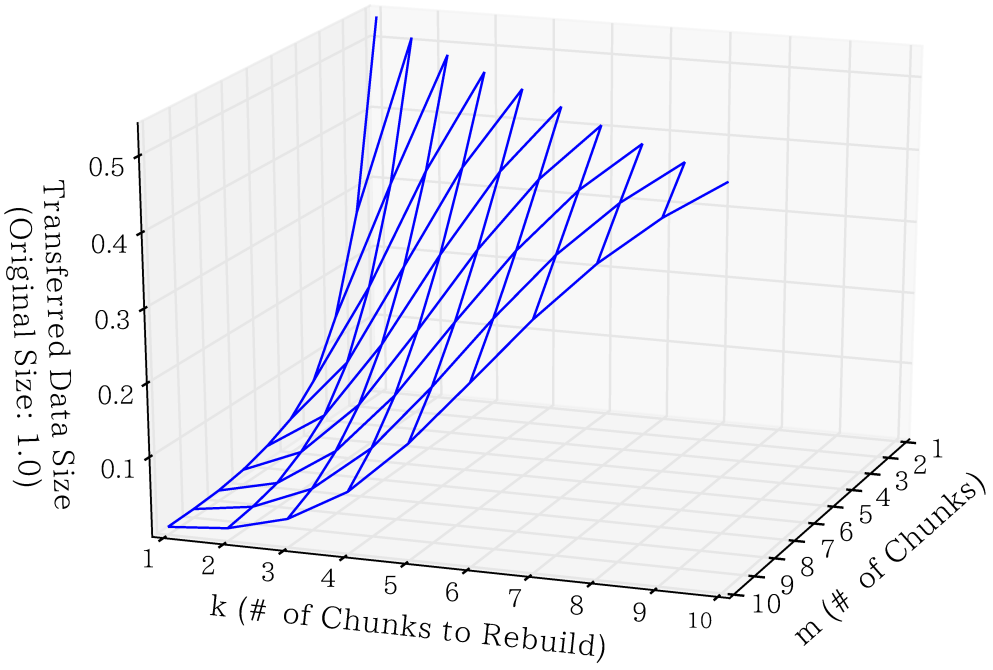
図 4.12: 各探索処理におけるデータセンタ間ホップ数と総ホップ数

すなわち，分散キーバリューストアの管理者はこれらのパラメータ変えることにより，ポリシーに応じた性能設定を行うことが可能である．

第二の実験では，他方のデータセンタに属するノード数の比率によってデータ



(a) ストレージ使用量



(b) `Get` 操作に伴うデータセンタ間の平均データ転送量

図 4.13: LDR におけるパラメータ設定の影響

センタ間の転送量がどのように変化するかを比較した．図 4.14 に結果を示す．各プロットは平均値，エラーバーは標準偏差を示す．他方のデータセンタに属するノードの比率が小さい場合は特に，データ転送量の平均値とばらつきが低減した．

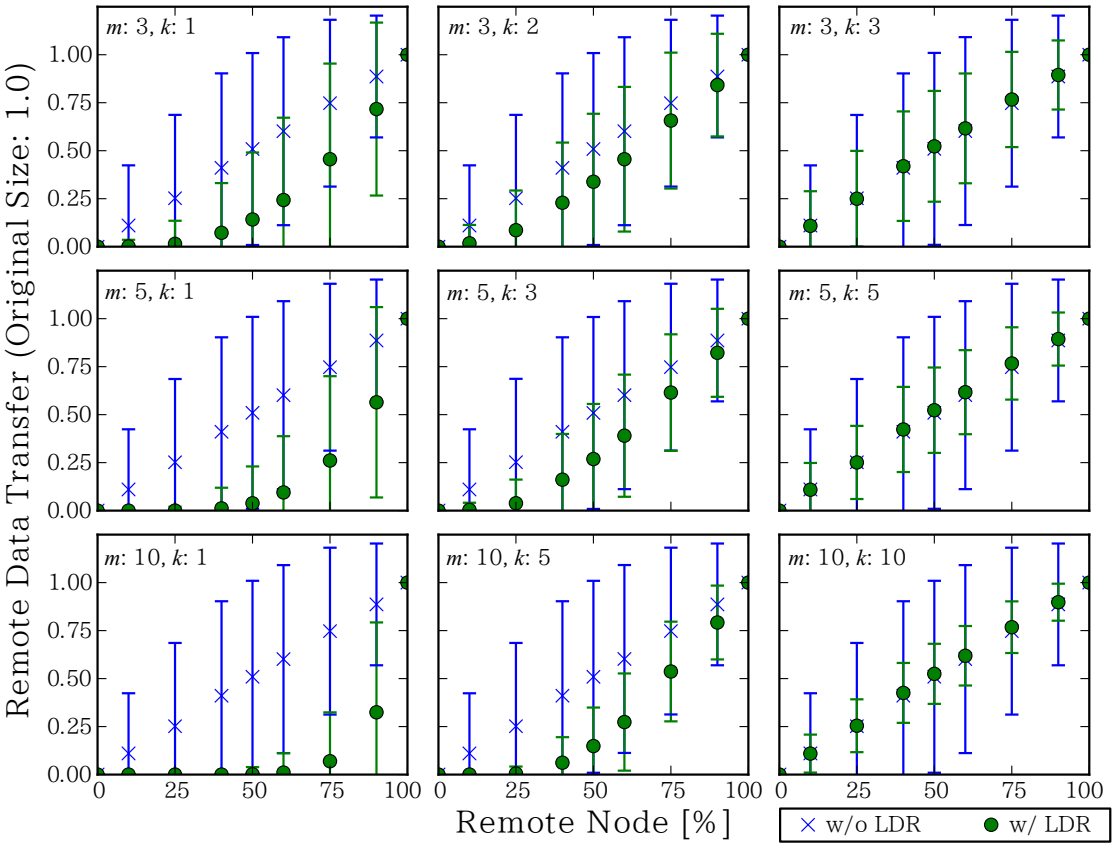


図 4.14: リモートデータセンタに属するノードの割合とデータ転送量の関係

これは LDR により，同じデータセンタに属するノードが保持するチャンクを優先的に利用され，他方のデータセンタからチャンクを取得する機会が減少するためである．また，パラメータ m, k がキーバリューストアの特徴に大きく影響した．パラメータ m はデータ転送量のばらつきに影響し，パラメータ k はデータ転送量の平均値に影響する．

4.6 議論

本節では，Pangaea の要素技術である ML-DHT のオーバヘッドついて定性的な議論を行う．

ML-DHT では各レイヤについて経路表を保持するため，レイヤの数が増加すると各ノードが保持する経路表のデータサイズも増加する．しかし，各レイヤに同じエントリが含まれる場合があるため，それらを統合することにより経路表全体のサイズは圧縮することが可能であると考えられる．

また，経路表のデータサイズが小さい DHT をベースとして，ML-DHT に拡張することでも，経路表サイズを抑えることが可能である．たとえば，拡張性の高い DHT アルゴリズムを構築する手法 Flexible Routing Tables (FRT) [139] は，各ノードの保持する経路表エントリを動的に整理することで，経路表エントリの増加を抑えつつ効率的な探索処理を実現する．FRT 自体には不必要なデータセンタ間通信を防ぐ仕組みは備わっていない．ML-DHT では基本となる DHT に FRT を用い ML-FRT とすることが可能であるため，補完的である．ML-DHT において各ノードは各レイヤについて経路表を保持するため，FRT を用いることで各ノードが保持する経路表をより削減することができる．

FRT にグループの概念を導入した GFRT [139] では，各ノードにグループを割り当て，同グループのノードが経路表に多く存在するよう整理することで，冗長なグループ間通信の発生を防ぐ．すなわち，各データセンタをグループとして扱うことで，冗長なデータセンタ間通信の発生を防ぐことができる．一方 ML-DHT は，各ノードがそれぞれ一つのグループに属する GFRT と異なり，複数の包含関係にあるグループを同時に扱うことで，冗長なデータセンタ間通信の発生を防ぐだけでなく探索状況に応じ段階的に探索範囲を拡大できる．例えば，図 4.6b に示したように，ML-DHT ではデータセンタ内の探索のみでは目的のノードに到達できない場合に，段階的に地域，全体と探索範囲を拡大していくことができる．このような探索は，図 4.4 のようにグループ自体の分布にも偏りがある場合，特に有効である．反対に，グループが均一に分布している状況下では，多数のレイヤ ($L > 2$) を用いた探索手法は特に効果的ではない．また，ML-DHT では各レイヤについて経路表を持つため，不必要に L を大きくすることは無用な経路表サイズの増大につながる．したがって，階層的な探索が不要で経路表のサイズを最小限にしたい場合は GFRT が適しており，階層的な探索が効果的な場合は ML-DHT が適している．

4.7 まとめ

本章では，サービスコアのストレージ層に着目した負荷分散手法 Pangaea について説明した．本手法は，複数のデータセンタを跨いだキーバリューストア構築し，どのデータセンタからでも同様のストレージアクセスを実現する．これにより，ウェブサーバやアプリケーションサーバはデータセンタの制約を受けず自由に配置

することが可能となり，高い拡張性および伸縮性を実現する．本手法は，狭帯域かつ高遅延であるデータセンタ間通信をデータセンタ間通信を減らす要素技術として，Multi-Layered Distributed Hash Table (ML-DHT) と Local-first Data Rebuilding (LDR) 用いる．これらの手法は共に，データセンタ間を跨ぐ分散キーバリューストアにおいてインターネットを介したデータセンタ間通信を減少させる働きをする．ML-DHT は，データセンタ間を跨ぐ通信が冗長に発生することを防ぐため，同じデータセンタに属するストレージサーバを優先的に探索し，データ探索時における通信遅延の増大を抑制する．LDR は，Erasure Coding を用いて保存するデータを冗長性を持ったチャンクに分割し，データセンタ間のデータ転送量の増大を抑制する．シミュレーション環境とインターネットを介した実環境を用いて行った提案手法の評価実験では，Chord を用いたシステムと比較してデータ探索に要する時間が 74 % 減少し，ストレージ使用量とデータ転送量を自由に調整できることを確認した．

第5章　結論

5.1　本研究のまとめ

近年，インターネット利用者数は増加し続けており，ウェブサービスの社会的重要度が高まっている．このような，利用者数の増加や重要性の高まりの一方で，ウェブサービスに対する Flash Crowds が問題となっている．Flash Crowds は，特定のサービスやコンテンツに対する突発的なアクセス集中である．Flash Crowds によってウェブサービスを提供するサーバが過負荷となると，サービスの性能低下や異常停止のような障害が発生する．そのため，ウェブサービスには Flash Crowds 対策が必要とされている．

Flash Crowds に対応するには，ウェブサービスを提供するサーバが過負荷となることを避ける必要がある．サーバが過負荷となることを避けるには，ウェブサービスに対する負荷を軽減および分散する必要がある．しかし，Flash Crowds は，発生の時期や規模の予測が困難であるため，予測を前提に適切な対策を講じることは難しい．したがって，Flash Crowds による需要変動に合わせ処理能力を適切に増減させる仕組みが Flash Crowds 対策として有効である．そのため，Flash Crowds 対策には 4 つの性質が求められる．第一に，投入した資源に応じて十分な処理能力を得られなければならない（拡張性）．第二に，Flash Crowds による負荷の増加よりも処理能力の増加が同等またはより速ければならない（敏捷性）．第三に，Flash Crowds が発生していない平常時などは適切な量の資源で運用できなければならない（伸縮性）．第四に，オリジナルコンテンツの更新内容が素早く反映されなければならない（一貫性）．

第 2 章で述べたように，既存手法は，1) プロキシ方式, 2) キャッシュ共有方式, 3) クラウド方式, 4) 完全複製方式 の 4 つの方式に大別される．しかし，いずれの方式についても，先に挙げた 4 つの要件をすべて満たすことは難しい．プロキシ方式は，静的に設置したプロキシサーバを用いるため，拡張性は高いが，敏捷性および伸縮性は低い．また，キャッシュがコンテンツ提供者の管理下にある場合

は一貫性は高い．キャッシュ共有方式は，クライアント間でキャッシュを共有させるため，負荷の集中点が生じない構造のものについては拡張性，敏捷性，伸縮性が高い．しかし，多数のキャッシュがクライアント間に散ってしまい，コンテンツの更新反映には時間を要するため，一貫性は低い．クラウド方式は，仮想化技術などを用いコンテンツ提供に使用する資源を動的に調整するため，拡張性は高い．敏捷性は，ウェブサービスのワークロード特性を熟知した者が適切にパラメータ設定した場合のみ高い．伸縮性は，データセンタ内で利用可能な資源の範囲内に限定され，これを超える場合には対応できない．また，データセンタ内の高速なネットワークで接続されたストレージを用いるため一貫性は高い．完全複製方式は，複数のデータセンタにサービスの完全な複製を配置するため，拡張性は高い．しかし，複製の設置には全データのコピーが必要となるため，敏捷性および伸縮性は低い．また，複製間の一貫性は，同期ポリシーに依存する．

本論文では，Flash Crowds による負荷を効果的に軽減および分散する手法を提案した．負荷軽減手法である MashCache は，クライアント間で構築するキャッシュ共有ネットワークにより，ウェブサービスに到達するリクエスト数自体を減少させる．本手法では，クライアントとキャッシュの管理に分散ハッシュ表 (DHT)，キャッシュ探索のキーに Query Origin Key を採用することで，管理用サーバなどの負荷の集中点が生じることを防ぎ，また，キャッシュの複製や分割をサポートする Cache Meta Data を用いることで各クライアントの負荷も分散するため，高い拡張性を実現する．また，クライアント資源を利用することにより，Flash Crowds による需要変動に追従して消費する資源が適切に伸縮するため，敏捷性および伸縮性も高い．すべてのコンテンツをキャッシュするという Aggressive Caching をキャッシュ生成ポリシーとすることで，どのコンテンツが Flash Crowds の対象となるかという予測を不要とする．さらに，Two-phase Delta Consistency による低負荷かつ高頻度なキャッシュ更新により，一貫性も高い．負荷分散手法である Pangaea は，データセンタ間を跨いぐ広域分散型キーバリューストアによって，ウェブサービスの 3 層構造におけるストレージ層の負荷分散を実現する．複製による負荷分散では無いため一貫性は高い．本手法では，任意のデータセンタに属するストレージサーバの資源を集約して単一キー空間の巨大なキーバリューストアを構築する．キー空間を単一とすることで各ストレージサーバに均一に負荷を分散することができるため，拡張性は高い．Consistent Hashing により，追加したノードへ移譲するデータ量が必要最低限であり，敏捷性は高い．利用可能な資源はデータセンタ内に用意された資源に限定されないため，伸縮性は高い．また，複製による負荷分散で

は無いため一貫性は高い．

提案手法をそれぞれ実装し，シミュレーション環境と実際のインターネット環境を用い有用性を評価した．MashCache が Flash Crowds を模したシミュレーションにおいて，ウェブサーバの負荷を約 98.2 % 軽減させることを確認した．ここで，各リクエストによって取得されたコンテンツのキャッシュがいつ生成されたものであるかを解析したところ，リクエストの 95 % 以上が過去 10 s より新しいことを確認した．また，Pangaea がデータセンタ間通信による性能低下を軽減し，ストレージの負荷分散手法として有用であることを確認するために，データの探索時間およびデータセンタ間のデータ転送量について評価したところ，それぞれ 74 %および 70 % 削減されることを確認した．

5.2 今後の展望

本論文では，Flash Crowds によるウェブサービスの障害に着目し，その対策として負荷軽減手法の MashCache および負荷分散手法の Pangaea を提案した．以下では，今後の研究の方向として課題について述べる．

本論文中で述べたように，Flash Crowds は発生の要因が多岐に渡るだけでなく，多くのクライアントが関わるなど複雑な事象である．そのため，Flash Crowds を予測することや，各クライアントに対してどのような品質でサービス提供できているかを正確に知ることは困難である．しかし，Flash Crowds に関わる事象を適切にモデル化することができれば，より効果的な対策手法が実現可能であると考えられる．

たとえば，サービス品質を保証した Flash Crowds 対策手法が挙げられる．商用のウェブサービスなどでは一定のサービス品質の保証を求められることが多いため，サービス障害の発生のみを避けることは十分ではない．したがって，Service Level Agreement (SLA) を満たすための方法について考慮する必要がある．SLA にはたとえば，応答遅延が挙げられる．MashCache において応答遅延を満たすには，キャッシュの探索に要する時間を見積もる仕組みが必要である．たとえば，通信遅延が一定値以下のクライアント同士をクラスタリングし，SLA を満たすクライアントからキャッシュを取得するといった手法が考えられる．Pangaea において応答遅延を満たすには，SLA で定められている応答遅延の厳しさに応じてデータを配置するデータセンタを選択する仕組みが必要である．たとえば，応答遅延が厳し

く設定されているデータオブジェクトについては複数のデータセンタに配置するなど，データオブジェクトごとに配置ポリシーを設定する手法が考えられる．

謝辞

本論文は，著者が慶應義塾大学大学院理工学研究科開放環境科学専攻の後期博士課程に在籍中の研究成果をまとめたものです．本研究を行うにあたり，また本論文をまとめるにあたり，多くの方々からご指導およびご協力を賜りました．この場を借りて，お世話になったすべての方々に対し御礼申し上げます．

まず，本論文の主査であり，著者の指導教員である慶應義塾大学理工学部情報工学科 河野健二教授に深く感謝いたします．河野健二教授には，著者が慶應義塾大学理工学部 4 年次に研究室配属されてから，同大学院修士課程および博士課程の計 7 年半もの長きにわたり，研究活動の意義，読者を納得させるための論文の執筆方法，聴衆に伝わるプレゼンテーションのまとめ方，そして何よりも研究活動の面白さをご教示いただきました．ここまで研究活動を進めることができたことは，ひとえに河野健二教授の親身なご指導のおかげです．また，国内外の様々な学会へ挑戦し，参加する機会を与えていただきました．さらに，高度な研究活動のために，国内有数の潤沢な研究設備を利用させていただきました．心より感謝いたします．

次に，本論文の副査をご担当いただいた，慶應義塾大学理工学部情報工学科 寺岡文男教授，金子晋丈専任講師，および同学部システムデザイン工学科 矢向高弘准教授に感謝いたします．副査の皆様には貴重なお時間を割いていただき，本論文を丁寧に査読していただきました．副査の皆様との有意義な議論により，本論文の完成度が大きく向上したと実感しております．深く感謝いたします．

東京農工大学先端情報科学部門 山田浩史准教授に感謝いたします．山田浩史准教授には，山田浩史准教授が博士課程の学生として河野研究室に在籍されていた頃から，研究を進めるにあたり様々な助言をいただきました．著者が博士課程への進学を考える最初のきっかけともなった存在であり，共に研究活動を行う機会を得たことは大変幸運でした．深く感謝いたします．

浅原理人博士に感謝いたします．浅原理人博士は，近い分野の研究をされていたこともあり，著者が研究室に配属され右も左もわからない頃より，研究を進め

る上で重要な助言をいくつもいただきました．心より感謝いたします．

糟谷正樹博士に感謝いたします．糟谷正樹博士とは，河野研究室への配属時より博士課程に至る計 6 年間の長きにわたり，同期として研究生活を送ってきました．思うような成果が出ない時期なども乗り越えることができたのは，共に博士課程に進学した同期の存在によるところが大きいと考えております．心より感謝いたします．

小菅祐史博士，殿崎俊太郎氏に感謝いたします．小菅祐史博士と殿崎俊太郎氏とは，著者が修士課程在籍中に，株式会社ライトマークスを共同で設立しました．今日まで研究活動と会社経営を並行して行うことができたことは，小菅祐史博士と殿崎俊太郎氏のご協力によるところが大きいと考えております．心より感謝致します．

慶應義塾大学理工学部情報工学科 河野研究室の皆様に感謝いたします．優秀な仲間に囲まれ刺激を受けながら研究を進めることができたことは大変幸せであると感じております．感謝いたします．

本論文の研究を進めるにあたり使用した実験機材の一部，本研究の原著論文の掲載，および国内外における発表につきましては，科学技術振興機構 CREST のご支援をいただきました．また，慶應義塾大学先端科学技術研究センターの KLL 後期博士課程研究助成金から本研究に対するご支援をいただきました．さらに，藤原奨学基金，慶応工学会，および日本学生支援機構による奨学金は，著者の研究活動の大きな支えとなりました．ここに感謝いたします．

最後に，様々な支援を惜しまず，著者の博士課程進学を支持し，現在まで暖かく見守っていただきました両親，両祖父母，そして弟たちに心より感謝いたします．

参考文献

[1] Jon Postel. Internet Protocol, 1981. RFC791.

[2] National Science Foundation. A Brief History of NSF and the Internet, 2003. https://www.nsf.gov/news/news_summ.jsp?cntn_id=103050, 最終アクセス: 2015 年 7 月 7 日.

[3] Miniwatts Marketing Group. World Internet Users and 2014 Population Stats. Technical report, 2014. http://www.internetworldstats.com/stats.htm, 最終アクセス: 2015 年 7 月 7 日.

[4] Ron Kohavi, Randal M. Henne, and Dan Sommerfield. Practical Guide to Controlled Experiments on the Web: Listen to Your Customers Not to the Hippo. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (*KDD '07*), pp. 959–967, 2007.

[5] Amazon.com, Inc. Amazon. http://www.amazon.com/, 最終アクセス: 2015 年 7 月 7 日.

[6] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kiciman, Matthew Merzbacher, David Oppenheimer, Naveen Sastry, William Tetzlaff, Jonathan Traupman, and Noah Treuhaft. Recovery Oriented Computing (ROC): Motivation, Definition, Techniques,. Technical report, 2002.

[7] David Patterson. A Simple Way to Estimate the Cost of Downtime. In *Proceedings of the 16th USENIX Conference on System Administration* (*LISA '02*), pp. 185–188, 2002.

[8] 株式会社 NTT ドコモ. 災害用伝言板. http://dengon.docomo.ne.jp/top.cgi, 最終アクセス: 2015 年 7 月 7 日.

[9] KDDI 株式会社. 災害用伝言板. http://dengon.ezweb.ne.jp/, 最終アクセス: 2015 年 7 月 7 日.

[10] 東日本電信電話株式会社. 災害用伝言板 (web171). https://www.web171.jp/, 最終アクセス: 2015 年 7 月 7 日.

[11] ソフトバンク株式会社. 災害用伝言板. http://dengon.softbank.ne.jp/, 最終アクセス: 2015 年 7 月 7 日.

[12] CERT Coordination Center. Denial of Service Attacks, 1999. http://www.cert.org/information-for/denial_of_service.cfm, 最終アクセス: 2015 年 7 月 7 日.

[13] Akamai Technologies. Akamai Provides Insight into Internet Denial of Service Attack. http://www-dualstack.akamai.com/html/about/press/releases/2004/press_061604.html, 最終アクセス: 2015 年 7 月 7 日.

[14] 浅原 理人. Flash Crowd による影響を軽減する複製サーバの配置先決定手法に関する研究. PhD thesis, 慶應義塾大学, 2010.

[15] SourceForge, Inc. Slashdot. http://slashdot.org/, 最終アクセス: 2015 年 7 月 7 日.

[16] Stephen Adler. The Slashdot Effect: An Analysis of Three Internet Publications. http://hup.hu/old/stuff/slashdotted/SlashDotEffect.html, 最終アクセス: 2015 年 7 月 7 日.

[17] Jaeyeon Jung, Balachander Krishnamurthy, and Michael Rabinovich. Flash Crowds and Denial of Service Attacks: Characterization and Implications for CDNs and Web Sites. In *Proceedings of the 11th ACM International World Wide Web Conference* (*WWW '02*), pp. 293–304, 2002.

[18] Facebook, Inc. Facebook. http://www.facebook.com/, 最終アクセス: 2015 年 7 月 7 日.

[19] Twitter, Inc. Twitter. http://twitter.com/, 最終アクセス: 2015 年 7 月 7 日.

[20] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the emerald system. *ACM Transactions on Computer Systems*, Vol. 6, No. 1, pp. 109–133, February 1988.

[21] Geoff C. Berry, Jeffrey S. Chase, Geoff A. Cohen, Landon P. Cox, and Amin Vahdat. Toward Automatic State Management for Dynamic Web Services. In *Proceedings of the 1999 Network Storage Symposium* (*Netstore*), 12 pages, October 1999.

[22] Germán Goldszmidt and Guerney Hunt. Scaling Internet Services by Dynamic Allocation of Connections. In *Proceedings of the 6th IFIP/IEEE International Symposium on Integrated Network Management* (*IM '99*), pp. 171–184, May 1999.

[23] Jim Challenger, Arun Iyengar, Karen Witting, Cameron Ferstat, and Paul Reed. A Publishing System for Efficiently Creating Dynamic Web Content. In *Proceedings of the 19th IEEE INFOCOM* (*INFOCOM 2000*), Vol. 2, pp. 844–853, March 2000.

[24] Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein, and David Culler. Scalable, distributed data structures for internet service construction. p. 14 pages, 2000.

[25] Valeria Cardellini, Emiliano Casalicchio, Michele Colajanni, and Philip S. Yu. The state of the art in locally distributed web-server systems. *ACM Computing Surveys*, Vol. 34, No. 2, pp. 263–311, June 2002.

[26] Haifeng Yu and Amin Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Transactions on Computer Systems*, Vol. 20, No. 3, pp. 239–282, August 2002.

[27] Cal Henderson. *Building Scalable Web Sites*. O'Reilly Media, 2006.

[28] Theo Schlossnagle. *Scalable Internet Architectures*. Sams Publishing, 2006.

[29] Jeremy Elson and Jon Howell. Handling Flash Crowds from Your Garage. In *Proceedings of the USENIX Annual Technical Conference* (*USENIX '08*), pp. 171–184, 2008.

[30] Google, Inc. Chrome. https://www.google.com/chrome/, 最終アクセス: 2015 年 7 月 7 日.

[31] Mozilla Firefox. https://www.mozilla.org/firefox/, 最終アクセス: 2015 年 7 月 7 日.

[32] Apple Inc. Safari. https://www.apple.com/safari/, 最終アクセス: 2015 年 7 月 7 日.

[33] Microsoft Corporation. Internet Explorer. http://www.microsoft.com/windows/internet-explorer/, 最終アクセス: 2015 年 7 月 7 日.

[34] Microsoft Corporation. Microsoft Edge. http://windows.microsoft.com/en-us/windows/preview-microsoft-edge-pc/, 最終アクセス: 2015 年 7 月 7 日.

[35] R. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content, 2014. RFC7230.

[36] R. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests, 2014. RFC7232.

[37] R. Fielding, Y. Lafon, and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Range Requests, 2014. RFC7233.

[38] R. Fielding, M. Nottingham, and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Caching, 2014. RFC7234.

[39] R. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Authentication, 2014. RFC7235.

[40] A. Barth. HTTP State Management Mechanism, 2011. RFC6265.

[41] The Apache Software Foundation. Apache HTTP Server. http://httpd.apache.org/, 最終アクセス: 2015 年 7 月 7 日.

[42] NGINX, Inc. Nginx. http://nginx.com/, 最終アクセス: 2015 年 7 月 7 日.

[43] Microsoft Corporation. Internet Information Services. https://www.iis.net/, 最終アクセス: 2015 年 7 月 7 日.

[44] Microsoft Corporation. ASP.NET MVC Framework. http://www.asp.net/mvc/mvc4, 最終アクセス: 2015 年 7 月 7 日.

[45] The Apache Software Foundation. Apache Struts. http://struts.apache.org/, 最終アクセス: 2015 年 7 月 7 日.

[46] Inc. Cake Software Foundation. CakePHP. http://cakephp.org/, 最終アクセス: 2015 年 7 月 7 日.

[47] Catalyst Foundation. Catalyst. http://www.catalystframework.org/, 最終アクセス: 2015 年 7 月 7 日.

[48] Django Software Foundation. Django. https://www.djangoproject.com/, 最終アクセス: 2015 年 7 月 7 日.

[49] Rails Core Team. Ruby on Rails. http://rubyonrails.org/, 最終アクセス: 2015 年 7 月 7 日.

[50] WSGI.org. Web Server Gateway Interface. http://wsgi.org/, 最終アクセス: 2015 年 7 月 7 日.

[51] D. Robinson and K. Coar. The Common Gateway Interface (CGI) Version 1.1, 2004. RFC3875.

[52] Inc. Sun Microsystems. NFS: Network File System Protocol Specification, 1989. RFC1094.

[53] B. Callaghan, B. Pawlowski, and P. Staubach. NFS Version 3 Protocol Specification, 1995. RFC1813.

[54] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. Network File System (NFS) version 4 Protocol, 2003. RFC3530.

[55] RedHat, Inc. Ceph. http://ceph.com/, 最終アクセス: 2015 年 7 月 7 日.

[56] Lustre. http://lustre.org/, 最終アクセス: 2015 年 7 月 7 日.

[57] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (*SOSP '03*), pp. 29–43, 2003.

[58] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *ACM SIGOPS Operating Systems Review*, Vol. 37, No. 5, pp. 29–43, October 2003.

[59] The Apache Software Foundation. Hadoop Distributed File System. `http://hadoop.apache.org/`, 最終アクセス: 2015 年 7 月 7 日.

[60] Edgar Frank Codd. A relational model of data for large shared data banks. *Communications of the ACM*, Vol. 13, No. 6, pp. 377–387, June 1970.

[61] Joint technical committee of the International Organization for Standardization (ISO) and the International Electrotechnical Commission. Information technology - Database languages - SQL, 1999. ISO/IEC 9075.

[62] Oracle Corporation. MySQL. `https://www.mysql.com/`, 最終アクセス: 2015 年 7 月 7 日.

[63] The PostgreSQL Global Development Group. PostgreSQL. `http://www.postgresql.org/`, 最終アクセス: 2015 年 7 月 7 日.

[64] Oracle Corporation. Oracle Database. `https://www.oracle.com/database/`, 最終アクセス: 2015 年 7 月 7 日.

[65] International Business Machines Corporation. IBM DB2 Database Software. `http://www-01.ibm.com/software/data/db2/`, 最終アクセス: 2015 年 7 月 7 日.

[66] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation* (*OSDI '06*), pp. 205–218, 2006.

[67] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems*, Vol. 26, No. 2, pp. 4:1–4:26, June 2008.

[68] The Apache Software Foundation. Apache HBase. `http://hbase.apache.org/`, 最終アクセス: 2015 年 7 月 7 日.

[69] Hypertable, Inc. Hypertable. `http://hypertable.org/`, 最終アクセス: 2015 年 7 月 7 日.

[70] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles* (*SOSP '07*), pp. 205–220, 2007.

[71] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. *ACM SIGOPS Operating Systems Review*, Vol. 41, No. 6, pp. 205–220, October 2007.

[72] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, Vol. 44, No. 2, pp. 35–40, April 2010.

[73] 10gen, Inc. MongoDB. `http://www.mongodb.org/`, 最終アクセス: 2015 年 7 月 7 日.

[74] Oracle Corporation. Oracle NoSQL Database. `http://www.oracle.com/us/products/database/nosql/`, 最終アクセス: 2015 年 7 月 7 日.

[75] Microsoft Corporation. Microsoft Azure Table Storage. `http://azure.microsoft.com/ja-jp/services/storage/tables/`, 最終アクセス: 2015 年 7 月 7 日.

[76] Salvatore Sanfilippo. Redis. http://redis.io/, 最終アクセス: 2015 年 7 月 7 日.

[77] Sanjay Ghemawat and Jeff Dean. LevelDB. https://github.com/google/leveldb, 最終アクセス: 2015 年 7 月 7 日.

[78] Fal Labs. Kyoto Cabinet. http://fallabs.com/kyotocabinet/, 最終アクセス: 2015 年 7 月 7 日.

[79] Michael J. Freedman. Experiences with CoralCDN: A Five-year Operational View. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation* (*NSDI '10*), 16 pages, 2010.

[80] Michael J. Freedman. The Coral Content Distribution Network. http://www.coralcdn.org/, 最終アクセス: 2015 年 7 月 7 日.

[81] Michael J. Freedman, Eric Freudenthal, and David Mazières. Democratizing Content Publication with Coral. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation* (*NSDI '04*), 14 pages, 2004.

[82] Reddit Inc. Reddit. http://www.reddit.com/, 最終アクセス: 2015 年 7 月 7 日.

[83] Amazon Web Services, Inc. Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2/, 最終アクセス: 2015 年 7 月 7 日.

[84] The Apache Software Foundation. CloudStack. https://cloudstack.apache.org/, 最終アクセス: 2015 年 7 月 7 日.

[85] Hewlett-Packard Development Company, L.P. HP Helion Eucalyptus. https://www.eucalyptus.com/, 最終アクセス: 2015 年 7 月 7 日.

[86] Google, Inc. Google App Engine. https://appengine.google.com/, 最終アクセス: 2015 年 7 月 7 日.

[87] Microsoft Corporation. Microsoft Azure. http://azure.microsoft.com/, 最終アクセス: 2015 年 7 月 7 日.

[88] salesforce.com, Inc. Heroku. https://www.heroku.com/, 最終アクセス: 2015 年 7 月 7 日.

[89] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (*SOSP '03*), pp. 164–177, 2003.

[90] Akamai Technologies, Inc. Akamai. http://www.akamai.com/, 最終アクセス: 2015 年 7 月 7 日.

[91] The Squid Software Foundation. Squid: Optimising Web Delivery. http://www.squid-cache.org/, 最終アクセス: 2015 年 7 月 7 日.

[92] Tyron Stading, Petros Maniatis, and Mary Baker. Peer-to-Peer Caching Schemes to Address Flash Crowds. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems* (*IPTPS '02*), pp. 203–213, 2002.

[93] Limelight Networks, Inc. Orchestrate Delivery Network. http://www.limelight.com/, 最終アクセス: 2015 年 7 月 7 日.

[94] Amazon Web Services, Inc. Amazon CloudFront. http://aws.amazon.com/cloudfront/, 最終アクセス: 2015 年 7 月 7 日.

[95] Petar Maymounkov and David Mazières. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems* (*IPTPS '02*), pp. 53–65, 2002.

[96] Michael J. Freedman and David Maziéres. Sloppy hashing and self-organizing clusters. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems* (*IPTPS '03*), pp. 45–55, 2003.

[97] XiaoYu Wang, WeeSiong Ng, BengChin Ooi, Kian-Lee Tan, and AoYing Zhou. BuddyWeb: A P2P-Based Collaborative Web Caching System. In *Proceedings of International Workshop on Peer-to-Peer Computing*, pp. 247–251, 2002.

[98] Cohen, Bram. Incentives Build Robustness in BitTorrent. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems* (*IPTPS '03*), 2003.

[99] Christos Gkantsidis, John Miller, and Pablo Rodriguez. Comprehensive View of a Live Network Coding P2P System. In *Proceedings of the ACM SIGCOMM Conference on Internet Measurement* (*IMC '06*), pp. 177–188, 2006.

[100] Mayur Deshpande, Abhishek Amit, Mason Chang, Nalini Venkatasubramanian, and Sharad Mehrotra. Flashback: A Peer-to-PeerWeb Server for Flash Crowds. In *Proceedings of the 27th IEEE International Conference on Distributed Computing Systems* (*ICDCS '07*), 8 pages, 2007.

[101] 佐藤良. P2P 通信技術: BitTorrent プロトコルを用いた大容量データ配信, 2009. CESA Developers Conference.

[102] R. Ahlswede, Ning Cai, S. Y.R. Li, and R. W. Yeung. Network Information Flow. *IEEE Information Theory*, Vol. 46, No. 4, pp. 1204–1216, September 2006.

[103] Sitaram Iyer, Antony Rowstron, and Peter Druschel. Squirrel: A Decentralized Peer-to-peer Web Cache. In *Proceedings of the 21st Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (*PODC '02*), pp. 213–222, 2002.

[104] Prakash Linga, Indranil Gupta, and Ken Birman. A Churn-resistant Peer-to-peer Web Caching System. In *Proceedings of the ACM Workshop on Survivable and Self-regenerative Systems* (*SSRS '03*), pp. 1–10, 2003.

[105] Benny Rochwerger, David Breitgand, Amir Epstein, David Hadas, Irit Loy, Kenneth Nagin, Johan Tordsson, Carmelo Ragusa, Massimo Villari, Stuart Clayman, Eliezer Levy, Alessandro Maraschini, Philippe Massonet, Henar Munoz, and Giovanni Tofetti. Reservoir - When One Cloud Is Not Enough. *Computer*, Vol. 44, No. 3, pp. 44–51, March 2011.

[106] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional Storage for Geo-replicated Systems. In *Proceedings of the 23st ACM Symposium on Operating Systems Principles* (*SOSP '11*), pp. 385–400, 2011.

[107] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage

with COPS. In *Proceedings of the 23st ACM Symposium on Operating Systems Principles* (*SOSP '11*), pp. 401–416, 2011.

[108] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-distributed Database. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation* (*OSDI '12*), pp. 251–264, 2012.

[109] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google&Rsquo;s Globally Distributed Database. *ACM Transactions on Computer Systems*, Vol. 31, No. 3, pp. 8:1–8:22, August 2013.

[110] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making Geo-replicated Systems Fast As Possible, Consistent when Necessary. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation* (*OSDI '12*), pp. 265–278, 2012.

[111] Kazuyuki Shudo. Overlay Weaver. `http://overlayweaver.sourceforge.net/`, 最終アクセス: 2015 年 7 月 7 日.

[112] 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.

[113] Shudo Kazuyuki, Tanaka Yoshio, and Sekiguchi Satoshi. Overlay Weaver: An overlay construction toolkit. *Computer Communications*, Vol. 31, No. 2, pp. 402–412, 2008.

[114] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (*SIGCOMM '01*), pp. 149–160, 2001.

[115] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. *ACM SIGCOMM Computer Communication Review*, Vol. 31, No. 4, pp. 149–160, August 2001.

[116] Ron Rivest. The MD5 Message-Digest Algorithm, 1992. RFC1321.

[117] Bernard Wong, Aleksandrs Slivkins, and Emin Gün Sirer. Meridian: A Lightweight Network Location Service Without Virtual Coordinates. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (*SIGCOMM '05*), pp. 85–96, 2005.

[118] Bernard Wong, Aleksandrs Slivkins, and Emin Gün Sirer. Meridian: A Lightweight Network Location Service Without Virtual Coordinates. *ACM SIGCOMM Computer Communication Review*, Vol. 35, No. 4, pp. 85–96, August 2005.

[119] Frank Dabek, Jinyang Li, Emil Sit, James Robertson, M. Frans Kaashoek, and Robert Morris. Designing a DHT for Low Latency and High Throughput. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation* (*NSDI '04*), pp. 7–7, 2004.

[120] Sebastian Kaune, Tobias Lauinger, Aleksandra Kovacevic, and Konstantin Pussep. Embracing the Peer Next Door: Proximity in Kademlia. In *Proceedings of the 8th International Conference on Peer-to-Peer Computing* (*P2P '08*), pp. 343–350, 2008.

[121] Thomas Locher, Stefan Schmid, and Roger Wattenhofer. eQuus: A Provably Robust and Locality-Aware Peer-to-Peer System. In *Proceedings of the 6th International Conference on Peer-to-Peer Computing* (*P2P '06*), pp. 3–11, 2006.

[122] Cisco Systems, Inc. Cisco Visual Networking Index: Forecast and Methodology, 2010-2015. Technical report, Cisco Systems, Inc., 06 2010.

[123] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proceedings of the 18th IEEE INFOCOM* (*INFOCOM '99*), Vol. 1, pp. 126 –134, Mar 1999.

[124] Pingdom AB. The REAL connection speeds for Internet users across the world (charts). Technical report, Pingdom AB, 11 2010. `http://royal.pingdom.com/2010/11/12/real-connection-speeds-for-internet-users-across-the-world/`, 最終アクセス: 2015 年 7 月 7 日.

[125] Enver Kayaaslan, B. Barla Cambazoglu, Roi Blanco, Flavio P. Junqueira, and Cevdet Aykanat. Energy-price-driven Query Processing in Multi-center Web Search Engines. In *Proceedings of the 34th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (*SIGIR '11*), pp. 983–992, 2011.

[126] Asfandyar Qureshi, Rick Weber, Hari Balakrishnan, John Guttag, and Bruce Maggs. Cutting the Electric Bill for Internet-scale Systems. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (*SIGCOMM '09*), pp. 123–134, 2009.

[127] Asfandyar Qureshi, Rick Weber, Hari Balakrishnan, John Guttag, and Bruce Maggs. Cutting the Electric Bill for Internet-scale Systems. *ACM SIGCOMM Computer Communication Review*, Vol. 39, No. 4, pp. 123–134, August 2009.

[128] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A Scalable Content-addressable Network. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (*SIGCOMM '01*), pp. 161–172, 2001.

[129] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A Scalable Content-addressable Network. *ACM SIGCOMM Computer Communication Review*, Vol. 31, No. 4, pp. 161–172, August 2001.

[130] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms* (*Middleware '01*), pp. 329–350, 2001.

[131] B. Y. Zhao, Ling Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A Resilient Global-scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, Vol. 22, No. 1, pp. 41–53, September 2006.

[132] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing* (*STOC '98*), pp. 654–663, 1997.

[133] Hakim Weatherspoon and John Kubiatowicz. Erasure Coding Vs. Replication: A Quantitative Comparison. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems* (*IPTPS '02*), pp. 328–338, 2002.

[134] Zooko Wilcox-O'Hearn. Zfec. `http://pypi.python.org/pypi/zfec/`, 最終アクセス: 2015 年 7 月 7 日.

[135] Amazon Web Services, Inc. Amazon Virtual Private Cloud. `http://aws.amazon.com/vpc/`, 最終アクセス: 2015 年 7 月 7 日.

[136] OpenVPN Technologies, Inc. OpenVPN. `https://openvpn.net/`, 最終アクセス: 2015 年 7 月 7 日.

[137] The Distributed Applications Support Team (DAST) at the National Laboratory for Applied Network Research (NLANR). iperf. `http://sourceforge.net/projects/iperf/`, 最終アクセス: 2015 年 7 月 7 日.

[138] William Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, Vol. 33, No. 6, pp. 668–676, June 1990.

[139] Hiroya Nagao and Kazuyuki Shudo. Flexible Routing Tables: Designing Routing Algorithms for Overlays Based on a Total Order on a Routing Table Set. In

Proceedings of the 11th International Conference on Peer-to-Peer Computing (*P2P '11*), pp. 72–81, 2011.

論文目録

定期刊行誌掲載論文

- 堀江 光, 浅原 理人, 山田 浩史, 河野 健二. データセンタ間通信による性能低下を抑えた広域分散型キーバリューストア構築手法. 情報処理学会論文誌コンピューティングシステム，Vol.8, No.2, pp.1–14, 2015 年 6 月.

- Hikaru Horie, Masato Asahara, Hiroshi Yamada and Kenji Kono. MashCache: Taming Flash Crowds by Using Their Good Features, *IPSJ Transactions on Advanced Computing System (ACS 37)*, Vol.5, No.2, pp.10–22, March 2012.

国際会議論文

- *Hikaru Horie, Masato Asahara, Hiroshi Yamada and Kenji Kono. Minimizing WAN Commiunications in Inter-Datacenter Key-Value Stores. In *Proceedings of the 7th IEEE International Conference on Cloud Computing (CLOUD '14)*, pp.490–497, June 2014.

- *Hiroshi Yamada, Takumi Sakamoto, Hikaru Horie and Kenji Kono. Request Dispatching for Cheap Energy Prices in Cloud Data Centers. In *Proceedings of the 2th IEEE International Conference on Cloud Networking (CloudNet '13)*, pp.210–213 November 2012.

- Takumi Sakamoto, Hiroshi Yamada, *Hikaru Horie and Kenji Kono. Energy-Price-Driven Request Dispatching for Cloud Data Centers. In *Proceedings of the 5th IEEE International Conference on Cloud Computing (CLOUD '12)*, pp.974–976, June 2012.

- *Hikaru Horie, Masato Asahara, Hiroshi Yamada and Kenji Kono. Inter-Datacenter Elastic Key-Value Storage. In *Proceedings of the 10th IEICE International Conference on Optical Internet (COIN '12)*, pp.71–72, May 2012.

国内学会発表

- *堀江 光, 浅原 理人, 山田 浩史, 河野 健二. データセンタ間通信による性能低下を抑えた広域分散型キーバリューストア構築手法. 第 26 回情報処理学会コンピュータシステム・シンポジウム (ComSys '14), pp.44-54, 2014 年 11 月.

- *堀江 光, 浅原 理人, 山田 浩史, 河野 健二. データセンタ間通信による性能低下を抑えたキーバリューストア構築手法. 第 128 回情報処理学会システムソフトウェアとオペレーティング・システム研究会報告, Vol.2014-OS-128, No.11, 10 pages, 2014 年 2 月.

- *白松 幸起, 堀江 光, 河野 健二. 需要の集中を考慮した分散 Key-Value Store におけるレプリカの動的配置手法. 第 128 回情報処理学会システムソフトウェアとオペレーティング・システム研究会報告, Vol.2014-OS-128, No.10, 8 pages, 2014 年 2 月.

- *堀江 光, 浅原 理人, 山田 浩史, 河野 健二. 複数のデータセンタを跨ぐ伸縮性を備えたキーバリューストレージの実現手法. 第 122 回情報処理学会システムソフトウェアとオペレーティング・システム研究会報告，Vol.2012-OS-122, No.7, 7 pages, 2012 年 8 月.

- *堀江 光, 浅原 理人, 山田 浩史, 河野 健二. MashCache: Flash Crowds 耐性を持つマッシュアップサービス実現手法. 第 116 回情報処理学会システムソフトウェアとオペレーティング・システム研究会報告, Vol.2011-OS-116, No.1, 9 pages, 2011 年 1 月.

- *堀江 光, 浅原 理人, 河野 健二. クライアント資源を利用した堅牢なマッシュアップサービスの実現手法. 第 112 回情報処理学会システムソフトウェアとオペレーティング・システム研究会報告，Vol.2009-OS-112, No.5, 8 pages, 2009 年 8 月.