2019 年度　卒業論文

ライブストリーミングに対応した分散ハッシュテーブルの検討

学籍番号: T18I917F
沈 嘉秋

指導教員: 萩原 威志 助教

新潟大学工学部情報工学科

Year 2019 Graduation thesis

Distributed hash table for live streaming

Student number: T18I917F
Yoshiaki Shin

Advising professor: Assistant Professor Takeshi Hagiwara

Department of Information Engineering, Faculty of Engineering,
Niigata University

概要

分散ハッシュテーブルはファイル共有サービス等に応用され、すでに普及している一方で、近年はブロックチェーンの関連技術としても注目を集めている。分散ハッシュテーブルの中でもKademlia はその実装の容易さとノードの出入りに対する耐性の強さから実用的なサービスへの応用が可能であり、多くのサービスで採用されている。近年、ライブストリーミングサービスの需要が高まっている一方で配信プラットフォームの事業者への依存が強くサービス利用者の立場は弱いものとなっている。そこで本論文では分散的なライブストリーミングサービスの基盤となるシステムを Kademlia を元に実装し、その評価を行った。Kademlia 上で単純にライブストリーミングの実装を行う場合、非効率的な通信が発生するが、本論文の手法では、Kademlia ネットワークを更に構造化することで非効率的な通信を削減することが可能となった。結果、分散的なライブストリーミングサービスの実装への足がかりを作ることが出来た。

キーワード　P2P, 分散ハッシュテーブル

abstract

While distributed hash tables have been widely used in file-sharing services, they have recently attracted attention as a blockchain-related technology. Among the distributed hash tables, Kademlia can be applied to practical services because of its ease of implementation and resistance to incoming and outgoing nodes, and it is used in many services. In recent years, while the demand for live streaming services has been increasing, there has been a strong dependence on providers of distribution platforms, and the position of service users has become weak. In this paper, we evaluate a system based on Kademlia, which is the basis of distributed live streaming services. When live streaming is simply implemented on Kademlia, inefficient communication occurs, but this paper can reduce inefficient communication by further structuring the Kademlia network. As a result, we were able to create a foothold for implementing a distributed live streaming service.

keywords P2P, distributed hash table

目次

第 1 章

序論

1.1 背景

近年、P2P ネットワーク技術がブロックチェーンなどによって再び注目を集めている。最近ではブロックチェーン流行以前に研究されていた P2P ネットワーク技術を利用した分散型ファイル共有サービスとブロックチェーンを組み合わせて開発されたサービスも登場している [1]。分散型ファイル共有サービスは分散ハッシュテーブルという技術を用いて実装されることがあり、分散型ファイル共有サービスの中でも利用者の特に多い BitTorrent[2] では Kademlia[3][4] という分散ハッシュテーブルを用いて開発されている。分散型ファイル共有サービスはこれまで、膨大なサーバリソースを持つ大企業などでなければ実現できなかった、大規模かつ、高速なファイル共有を実現した。

近年の家庭用ネットワークやモバイルネットワークの環境改善に伴い Youtube Live[*1] や Twitch[*2] といった高いスループットが要求される動画ライブストリーミングサービスが急速に普及してきている。しかし、これらのサービスは膨大なサーバリソースを持つ大企業や大企業の提供するクラウド環境を利用するなどしなければ実現できず、プラットフォーム事業者やクラウド事業者への依存が強く、サービス利用者の立場は弱いものとなっており不健全な状態にある。

1.2 目的

分散ハッシュテーブルの中でも Kademlia はその実装の容易さと高い churn 耐性[*3]を持つことから実用的なサービスへの応用が可能である。しかし、その一方で Kademlia 上で単純にライブストリーミングの実装を行う場合、高頻度に追加されるストリームのチャンクデータの共有をスケールさせることは困難である。

そこで本論文では、分散型の動画ライブストリーミングサービスの基盤となるシステムを、

[*1] https://www.youtube.com/live
[*2] https://www.twitch.tv
[*3] ノードの出入りに対する耐性の強さ

Kademlia をベースに実装する。成果物が Web ブラウザと Node.js で動作するライブラリとなるように開発を行う。

成果物ができるだけ多くのプラットフォームで動作するようにするために P2P 通信箇所に WebRTC [5] を用いる。WebRTC は Web ブラウザなどといったフロントエンド環境とサーバーサイド環境の両方に対応した低遅延通信のための規格である。また WebRTC は近年では、スマートフォンやパーソナルコンピュータに搭載されている Web ブラウザで動作する唯一の P2P 通信規格でもある。

完成したライブラリを用いて Web ブラウザ上で動作する分散ライブ動画配信アプリのサンプルを開発し動作確認を行う。完成したライブラリを用いたベンチマークプログラムと一般的な Kademlia を用いたベンチマークプログラムを Node.js 上で実行しその性能や性質の比較を行い有用性などの検証を行う。

1.3 本論文の構成

本論文では TypeScript[*4] というプログラム言語を用いて研究を行っている。そのため、本文中に登場するコードサンプルはすべて TypeScript によるものである。

[*4] https://www.typescriptlang.org

第 2 章

関連事例

2.1 分散ハッシュテーブル

分散ハッシュテーブルとは、分散型の key-value ストアを実現する手法であり、あるデータとそのハッシュ値をペアとしたハッシュテーブルを P2P ネットワーク上で複数のノードによって分散的に実装する技術である。複数のノードにデータを分散配置を行うため適切な構造化を行う必要がある。構造化には様々な手法が存在し、Chord や Kademlia といったさまざまな実装が存在する。分散ハッシュテーブルの実装の優劣はデータの探索効率、Churn 耐性、実装の容易さなどによって付けられる。

2.2 Kademlia

Kademlia は分散ハッシュテーブルの一種である。高い Churn 耐性を持つため、実用的な P2P アプリケーションにて多く利用されている。Kademlia はノード数 N のシステムにおいてデータを探索する際に $O(\log(n))$ 回ノードへの通信を行う。

本研究では Kademlia を TypeScript で実装した。Node.js と Chrome 上で動作するようにするために P2P 通信部分に WebRTC を利用した。

2.2.1 採用例

Kademlia は高い Churn 耐性を持ちながら、実装も容易であるため、多くの分散型のサービスで利用されている。ここでは、Kademlia を利用している有名なサービスを幾つか紹介する。サービスの紹介を表 2.1 に示す。

表 2.1　Kademlia を利用したサービス

サービス名	使用箇所
Torrent	magnetURL という機能を用いてファイルをダウンロードする際に 目的のファイルを持っているノードを探索するのに Kademlia を用いている。 Torrent はアクティブユーザと転送量という点で見ると 世界で最も成功した P2P のシステムであり、 そのシステムに Kademlia はおおいに貢献していると言える。
Ethereum	Node Discovery Protocol v4 というノードの探索プロトコルに用いられている.
IPFS	IPFS とは複数のノードが協調して一つの大きなストレージ または HTTP の置き換えとして機能することを目的としているシステムある。 IPFS は Kademlia をベースとして開発されている

2.2.2　Kademlia のアルゴリズム

ノード ID とキー

Kademlia では個々のノードに固有のノード ID が割り振られており、この ID を元にルーティングを行う。このノード ID は 160bit と定義されている。ノード ID の決定方法は、ランダムな値に sha1 というハッシュ関数を適用し、160bit の値を取り出すのが一般的である。また、ハッシュテーブルに保存するバリューと対になるキーも 160bit と定義されている。

経路表

分散ハッシュテーブルのアルゴリズムによってノードを管理する経路表の形は様々である。例えば、Chord という分散ハッシュテーブルの場合は環状の経路表を持っている。Kademlia は k-buckets という 160 個の k-bucket からなる経路表を持っている。一つの k-bucket には K 個 (たいていの実装例では 20 個) のノードが登録でき、自身のノードとの距離に応じた k-bucket にそれぞれのノードが登録されていく。ノード間の距離は 2 進数のノード ID 同士を XOR で掛け合わした結果を 10 進数に戻した値を用いる。

プロトコル

Kademlia には 4 種類の通信問い合わせがある。名称と内容についてまとめる。

- PING
  対象のノードがオンラインかどうかを問い合わせる。

- STORE
  対象ノードに key,value の組を保持させる。保持させる際のルールは、自身の k-buckets

から最も key に xor の距離が近いノードを選択し、そのノードに key,value を与える。受け取ったノードは更に自身の k-buckets から受け取った key に最も近いノードを選択し、key,value を与える。この動作を何度も繰り返す。最終的にはネットワーク上で最も key に距離が近いノードが目的の key,value を持つ。

- FIND_NODE
  自身の k-buckets のうち最も key に xor 距離の近いノードに自身のノード ID と距離が近い上位 K 個のノードの情報を送らせる。

- FIND_VALU
  自身の k-buckets のうち最も key に xor 距離の近いノードに key と対応する value を持っているか問い合わせる。持っている場合はその value を、持っていない場合は問い合わせられたノード自身の k-buckets のうち、key に最も近いノードの情報を返す。

ノードの管理

Kademlia の Churn 耐性の高さはこのノードの管理方法にある。Kademlia のノード管理は上記の 4 つのプロトコルの通信を行うついでに経路表を更新することで行われる。そのため、ノードの離脱の際の処理が必要なく、ノードの離脱を考慮することなくネットワークを維持することができる。

経路表の更新

ノードは 4 つのプロトコルのいずれかのメッセージを受け取った際に送信元が該当する k-bucket の中にあった場合そのノードを k-bucket の末尾に移す。送信元が該当する k-bucket の中に存在しないせず、k-bucket がすでに満杯な場合、その k-bucket 中の先頭のノードがオンラインかどうかを PING で確認する。オンラインなら先頭のノードを残し、そうでなければ送信元の新しいノードを k-bucket に追加する。こうすることで長時間オンラインになっているノードが優先的に k-bucket に残るため、ネットワークの安定性が増す。

ノードの新規参加

新規参加するノードは、まず接続先のノードに対して自身のノード ID を key として FIND_NODE を行う。問い合わせを受けたノードは送信元の key に近い最大 K 個のノードの情報を送信元に STORE する。そうすることで、新規参加するノードはまず最大 K 個のノードに接続される。このあと、さらに自身の k-buckets のうち最も自身のノード ID に xor 距離が近いノードに対し自身のノード ID を key とした FIND_NODE を繰り返すことで接続先のノードを増やすことができる。

ノードの離脱
　何もしない

2.3　WebRTC

本論文では、ブラウザとネイティブ環境の両方で動作する P2P 通信手法が要求される。そこで, その要求を満たす、WebRTC を P2P 通信部分に使用した。

WebRTC とは W3C が提唱するリアルタイム通信用の規格で，プラグイン無しでウェブブラウザ間のボイスチャット，ビデオチャット，ファイル共有ができる．WebRTC はブラウザ向けの規格として誕生したが，現在では，Android や iOS といったネイティブ環境で実装するためのライブラリが公開されている．サーバサイド向けにも libwebrtc[*1] などの実装が存在する。WebRTC は Google によってオープンソース化されており，現在は，W3C によってブラウザ対応 API の標準化が進められている。WebRTC には NAT 越えを実現するために ICE[6] という仕組みを採用している。

2.3.1　NAT 越え

NAT とは，インターネットプロトコルによって構築されたコンピュータネットワークにおいて，パケットヘッダに含まれる IP アドレスを，別の IP アドレスに変換する技術である．プライベートネットワーク環境下でプライベート IP アドレスを持つホストから，グローバル IP アドレスを持つゲートウェイを通して，インターネットにアクセスする際に，プライベート IP アドレスをグローバル IP アドレスに変換するために利用されることが多い．モバイルネットワークにおいてはキャリアグレード NAT が用いられている．そのため，スマートフォン間で P2P 通信を行うためには，NAT 越えを行う必要がある．本研究では WebRTC を用いて NAT 越えを行う．WebRTC では ICE の情報をやり取りすることで，NAT 越えを行っている．ICE とは通信可能性のある通信経路に関する情報を示し，文字列で表現される．次のような複数の経路を候補とする．

・P2P による直接通信
・STUN による，NAT 通過のためのポートマッピング
・TURN による，リレーサーバーを介した中継通信

STUN[7] とは，P2P 通信を行うアプリケーションにおいて，NAT 越えの方法の 1 つとして使われる標準化されたインターネットプロトコルである．STUN プロトコルは，アプリケーションが NAT の存在と種類を発見し，リモートホストへの UDP 接続に NAT が割り当てたグローバル IP アドレスとポート番号とを得ることを許す．STUN プロトコルが動作するには，インターネッ

[*1] https://webrtc.googlesource.com/src

ト上に STUN サーバが存在する必要がある．

TURN[8] とは，NAT やファイアウォールを超えた通信することを補助するためのインターネットプロトコルである．TURN が一番役立つのは，TCP， UDP を使って対象型 NAT 装置により隠蔽されたプライベートネットワークに接続されたクライアントで利用する場合である．

本研究では Google が無料公開している STUN サーバ[*2] を用いている．本研究では TURN サーバが無くとも P2P 通信が疎通する環境で検証を行うため、TURN サーバは利用していない。

2.3.2 シグナリング

WebRTC では，SDP と ICE Candidate の二つの情報を端末間で交換することによって P2P 通信が開始される．この SDP 等を交換する作業をシグナリングと言う．シグナリングを行うためには SDP 等を交換する必要がある．シグナリングには Trickle Ice と Vanilla Ice の 2 つの方法がある。本研究では Vanilla Ice というシグナリング手法を用いる．Vanilla Ice は，実装が容易である，シグナリングサーバとの通信回数が少ないというメリットがある一方，P2P 接続の完了にかかる時間が Trickle Ice より長くなる傾向がある．Vanilla Ice の手順について説明する．Vanilla Ice の概要図を図 2.1 に示す．

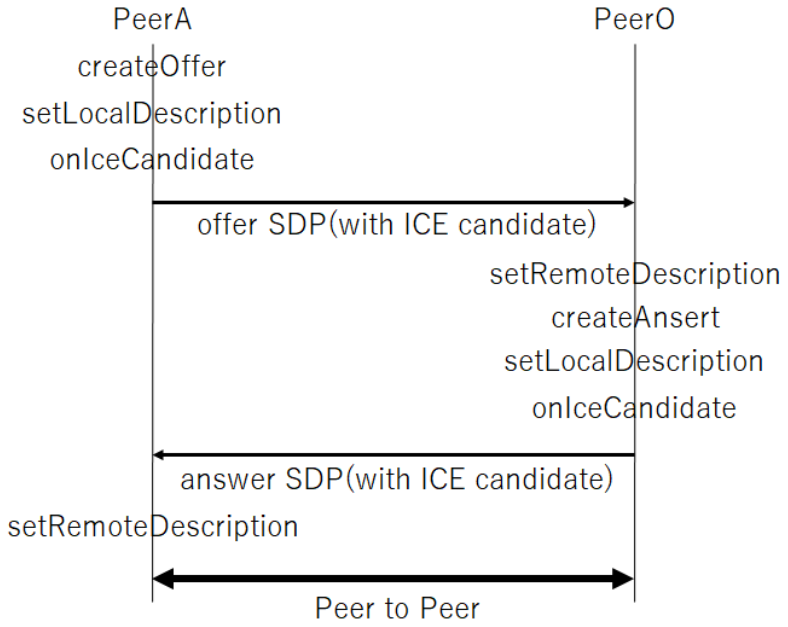

図 2.1 シグナリング

PeerA が createOffer を行い offer 側の SDP の作成準備を行う．次に setLocalDescription で SDP を作成し，ICE candidate のリストアップを行う．ICE candidate のリストアップが完了

[*2] stun.l.google.com:19302

すると，ICE candidate を offer 側の SDP の中に含ませて，PeerO へ送る．PeerO は PeerA の offer 側の SDP を setRemoteDescription で受け取り，createAnswer で answer 側の SDP の作成準備を行う．setLocalDescription で SDP を作成し，ICE candidate のリストアップを行う．ICE candidate のリストアップが完了すると ICE candidate を answer 側の SDP の中に含ませて，PeerA へ送る．PeerA は PeerO の answer 側の SDP を setRemoteDescription で受け取り P2P 接続が完了する．

第 3 章

提案手法

3.1 概要

本章では、Kademlia をベースにネットワークの構造化を行うことによって効率的にストリーム形式のデータを共有できるようにするシステムである LayeredKad を提案する。LayeredKad が分散型のライブストリーミングサービスの開発基盤になれるような性能を発揮できることを目指す。

Kademlia はデータを共有する際に、そのデータをアルゴリズムに従いネットワーク全体に分散させる。そのため Kademlia で連続的なストリーム形式のデータを共有する場合、個々のストリームのチャンクデータをネットワーク全体に対し問い合わせを行い、処理する必要がある。ライブ映像のようなチャンクの生成周期が非常に短く、高頻度にデータ共有する必要がある場合、Kademlia では、ネットワーク全体に対して連続的にその都度、負荷をかけることになり、データ共有の効率が非常に悪化することが予測される。

そこで LayeredKad では共有するデータごとに別の Kademlia ネットワークを作成しネットワーク自体のスコープをデータごとに区切ることで、データの共有がネットワーク全体への負荷にならないようにし、ストリームデータのような連続的なデータでも効率よく共有できるようにする。

本手法では、共有するデータの情報をメタデータとして、Kademlia ネットワーク上で共有する。このメタデータを共有するネットワークを MainNet と定義する。MainNet 上でメタデータの実体データの共有を目的とするユーザ同士でさらに別の Kademlia ネットワークを構築し、メタデータの情報に従いメタデータに対応する実体データの共有を行う。この実体データを共有するネットワークを SubNet とする。MainNet と SubNet の関係を図 3.1 に示す。

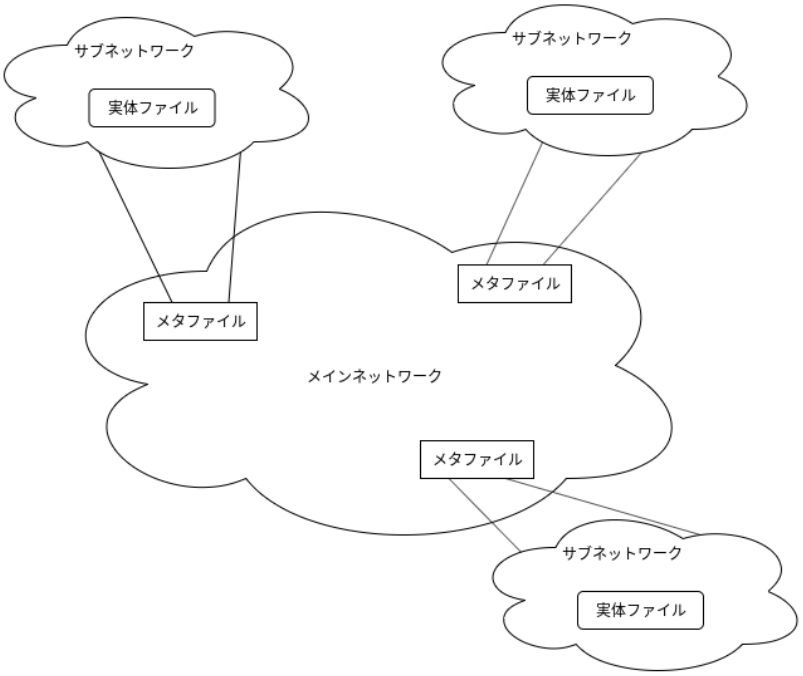

図 3.1　MainNet と SubNet

3.2　メタデータ

LayeredKad では静的、動的の両方のデータ形式の共有に対応するためにメタデータとして共有するデータの情報の共有を行い、メタデータの情報に従って振る舞いを変化させる。メタデータの基本構造は次のようになっている。

```
type Meta = {
  type: "static" | "stream";
  name: string;
  payload: { [key: string]: any };
};
```

type はメタデータの種類を表す。本論文では static と stream の二種類が存在する。
name はメタデータの名前である。name には任意の文字列を与える。
payload には実体データに関する情報を与える。

3.2.1　StaticMeta

静的なデータを扱うメタデータ

```
type StaticMeta = Meta & {
  type: "static";
```

```
  payload: { keys: string[] };
};
```

payload の keys が実体データのハッシュキーの集合である。

3.2.2 StreamMeta

ストリーム (ライブ映像など) を扱うメタデータ

```
type StreamMetaPayload = {
  first: string;
  width?: number;
  height?: number;
  cycle: number;
};
type StreamMeta = Meta & {
  type: "stream";
  payload: StreamMetaPayload;
};
```

payload の first が実体データのストリームデータの最初のチャンクのハッシュキーである。
width,height は動画の縦横のピクセル数である。
cycle には ”1000 ÷ フレームレート” の値を与える.

ストリームデータ

ストリームデータは複数の動的に生成されるチャンクデータから成り立つ。本論文では一つのキー (最初のチャンクのハッシュキー) から全チャンクデータを探索できるようにする必要がある。そのため、チャンクデータを Kademlia 上に保存する際には、そのチャンクデータ (便宜上 chunks とする) の次に生成されるチャンクデータ (便宜上 next chunks とする) が生成されるのを待ち、chunks と next chunks のハッシュキーを一つの value としてに Kademlia 上に保存し、next chunks のハッシュキーを辿ることで、ストリームデータを動的に継続的に取得できるようにしている。

3.3 Network

LayeredKad ではネットワークが MainNet と SubNet の 2 階層存在する。

3.3.1 MainNet

Kademlia をベースとしたネットワークである。ここでメタデータのやり取りと、新規参加ノードと SubNet との橋渡しを行う。

ノードの種類

LayeredKad はウェブブラウザのような公開されたトランスポートアドレス[*1]を持たないクライアントデバイスとサーバのような公開トランスポートアドレスを持つデバイスの両方で動作することを目的としている。ここでは公開トランスポートアドレスを持たないノードを GuestNode と呼び、公開 I トランスポートアドレスを持つノードを PortalNode と呼ぶ。

ノードの接続方法

LayeredKad の MainNet ではノード間は最終的には WebRTC で通信を行うが、WebRTC は通信を開始するためにシグナリングを行う必要がある。シグナリングのパターンは PortalNode と PortalNode、GuestNode と PortalNode、GuestNode と GuestNode の 3 パターンを考慮する必要がある。

- PortalNode と PortalNode
  Portal ノード間はトランスポートアドレスが公開されているので、相手のノードのトランスポートアドレスを知っている前提で、http[9] で接続を行い、http を経由して Vanilla Ice でシグナリングを行い、WebRTC の通信を開始する。WebRTC の peer を Kademlia の k-buckets で管理し、Kademlia ネットワークを構築する。

- GuestNode と PortalNode
  GuestNode は公開トランスポートアドレスを持たないので、必ず GuestNode から PortalNode へ接続を要求する必要がある。(逆は不可能) GuestNode は PortalNode のトランスポートアドレスを知っている前提で http で接続を行い、http を経由して Vanilla Ice でシグナリングを行い、WebRTC の通信を開始する。WebRTC の peer を Kademlia の k-buckets で管理し、Kademlia ネットワークを構築する。

- GuestNode と GuestNode
  GuestNode は公開トランスポートアドレスを持たないので,GuestNode 同士で独立して接続を開始することは出来ない。そのため、GuestNode 同士が接続されるパターンとして MainNet に WebRTC での接続を開始 (PortalNode + PortalNode か GuestNode + PortalNode のパターンで) した後に Kademlia のアルゴリズムに従い、接続されるケースが想定される。

ノード間の接続完了後はネットワーク全体で WebRTC を用いて共通のプロトコルで通信するので、あとは Kademlia の仕組みに従う。

[*1] IP アドレスとポートのペア

MainNet の命令

MainNet には Store,FindValue,DeleteValue の 3 つの命令が定義されている。

- Store
  MainNet 上にメタデータを保存する命令である。

  Store の型定義を以下に示す。

```
1        type Store = (meta: Meta) =>
2                Promise<{ url: string, peers: Peer[] }>
```

  Store はメタデータを引数として実行する。実行結果としてメタデータが保存されたアドレス (url) とメタデータを受け取ったノードらの情報 (peers) を返す。

- FindValue
  url に対応するメタデータを探す命令である。

  FindValue の型定義を以下に示す。

```
1        type Store = (url: string) =>
2                Promise<{ meta: Meta, peer: Peer }>
```

  FindValue は url 文字列を引数として実行する。実行結果としてメタデータ (meta) と、そのメタデータを返却してきたノードの情報 (peer) を返す。

- DeleteData
  自身の持つメタデータを削除する命令である。

  DeleteData の型定義を以下に示す。

```
1        type Store = (url: string) => void
```

  DeleteData は url 文字列を引数として実行する.

3.3.2 SubNet

Kademlia をベースとしたネットワークである。メタデータの指し示すデータのやり取りを行う。構造としては、一つの MainNet 上に複数の SubNet が存在することになる。

ノードの種類

SubNet は MainNet 上で動作するのですべての通信に WebRTC を使用している。そのため MainNet と違い、ノードの種類は 1 種類しか無い

SubNet の命令

SubNet には FindStaticMetaTarget と FindStreamMetaTarget の 2 種類の命令が定義されている。

- FindStaticMetaTarget
  静的なデータを扱うメタデータの実体データを探索する命令である。

  型定義を以下に示す。

```
type FindStaticMetaTarget = () => Promise<ArrayBuffer |
    undefined>
```

  SubNet の持つメタデータを元に実体データを探索し、探索結果を返却する。

- FindStreamMetaTarget
  StreamMeta の実体データを探索する命令である。

  型定義を以下に示す。

```
type FindStaticMetaTarget = (
        cb: (res: {
                        type: "error" | "chunk" | "complete";
                        chunk?: ArrayBuffer;
                        }) => void
        ) => void
```

  ストリームデータは長さが不明なので終了を持ち受けるのではなく、入手したチャンクを都度コールバックで返却する。

3.4 Actor

Actor は MainNet と SubNet を利用し、LayeredKad のシステムを実現するためのノードの実装である。本論文では Seeder,User,Navigator の 3 つの Actor が存在する。一つのノードは同時に複数の Actor になりうる。

3.4.1 Seeder

Seeder とはメタデータの実体データを持つノードのことである。Seeder の中でも特に任意のデータをもとにメタデータを作成し、メタデータを MainNet 上に Store し、SubNet を生成する Seeder を OriginSeeder と定義するが、振る舞いは通常の Seeder と変わらない。Seeder によって MainNet 上でメタデータを Store されたノードは Navigator となり、SubNet への接続を要求する User と Seeder の橋渡しを担う。また、Seeder も Navigator として振る舞う。

SubNet の生成

OriginSeeder はメタデータを MainNet に Store し、その結果メタデータの url と Store 先のノード情報 (peers) を得る。OriginSeeder はメタデータの情報を持った SubNet を生成する。そして peers の対象のノードたちを Navigator にする。

Seeder の命令

Seeder には OriginSeeder 用の 2 種類の命令が定義されている。

- StoreStatic
  静的なデータからメタデータを生成し MainNet に保存する。その後メタデータに対応する SubNet を生成し、メタデータの実体データを SubNet に保存する。
  型定義を以下に示す。

```
type StoreStatic = (name: string, ab: Buffer) =>
        Promise<{url: string, meta: Meta}>
```

  引数にメタデータの名前となる文字列と静的なデータを受け取る。メタデータの url とメタデータを返却する。

- StoreStream
  ストリームデータからメタデータを生成し MainNet に保存する。その後メタデータに対応する SubNet を生成し、メタデータの実体データを SubNet に保存する。
  型定義を以下に示す。

```
type StoreStream = (name: string, first: Buffer,
        payload:{width:number,heigh:number,cycle:number}
                ) =>
        Promise<{event:Trigger,url:string}>
```

  引数にメタデータの名前となる文字列とストリームデータの最初のチャンクデータとストリームデータの情報を受け取る。戻り値の Trigger は first チャンク以降のチャンクデータを送るためのコンポーネントである。url はメタデータの url である。

3.4.2 User

User はメタデータの URL をもとに MainNet 上でメタデータを探索し、メタデータに対応する SubNet と接続する。メタデータを持つ Navigator を仲介として Seeder と接続する。Seeder と User のピアは SubNet 上で管理される。SubNet と接続している User を特に Observer と呼ぶ

User の命令

Seeder には 1 種類の命令が定義されている。

- ConnectSubNet
  url を元に MainNet 上でメタデータを探索し、メタデータを与えてくれたノードを Navigator としてメタデータの実体データを持つ、Seeder の SubNet へ接続する。

  型定義を以下に示す。

```
1        type ConnectSubNet = (url: string) =>
2                Promise<{subNet: SubNet,meta:Meta}>
```

  メタデータの url を受け取り、SubNet とメタデータを返却する。

User は ConnectSubNet によって接続した SubNet に対して FindStaticMetaTarget や FindStreamMetaTarget を行うことによってメタデータの実体データを探索し入手する。

3.4.3 Navigator

Navigator は User を MainNet と SubNet 間の橋渡しを行う Actor である。MainNet を監視しており、User から ConnectSubNet 命令を受け取った時に、Navigator と接続された Seeder と User の接続を仲介する。

Seeder との接続

LayeredKad のノードはメタデータを Store されると、Navigator となるので、すべてのノードが潜在的に Navigator になるうる。そこで、これから Navigator になるノードのことを NavigatorCandidate と定義する。NavigatorCandidate は MainNet を監視しており、自らにメタデータを Store された時に、Store してきたノード (Seeder) に対して peer を生成しコネクションを維持する。これにより NavigatorCandidate は Navigator となる。

User と Seeder の接続仲介

Navigator は MainNet を監視し、User が自らの持つメタデータを探索によって発見し、SubNet への接続要求である。ConnectSubNet 命令を実行した際に自らの接続する Seeder に対して User

との接続を行うための情報を要求する (WebRTC の SDP 等) その情報を User へ返却し、次に User の接続情報 (SDP 等) を Seeder へ渡す。これにより User と Seeder の間に peer が生成され接続が完了する。Seeder は SubNet 環境下に存在するので、User はメタデータに対応する SubNet に接続できたことになる。

3.5 動作

LayeredKad 上で実際にデータを共有するケースの流れについて確認し、LayeredKad の動作を確認する。

3.5.1 web ブラウザから静的なデータを共有する

静的データの保存

1. web ブラウザは GuestNode にあたるので、Portal ノードのトランスポートアドレスを何らかの方法で入手し、PortalNode と接続し、MainNet にアクセスする

2. web ブラウザは Actor の Seeder として StoreStatic 命令を実行し、メタデータを MainNet に保存し、SubNet を生成し、Navigator と接続する

静的データの探索

1. web ブラウザは GuestNode にあたるので、Portal ノードのトランスポートアドレスを何らかの方法で入手し、PortalNode と接続し、MainNet にアクセスする

2. web ブラウザは Actor の User として ConnectSubNet 命令を実行し、Navigator を経由して Seeder の SubNet へ接続する。

3. web ブラウザは SubNet の FindStaticMetaTarget 命令を実行し、SubNet 上でメタデータの実体データを探索し、入手する。

3.5.2 web ブラウザかストリームデータを共有する

ストリームデータの保存

1. web ブラウザは GuestNode にあたるので、Portal ノードのトランスポートアドレスを何らかの方法で入手し、PortalNode と接続し、MainNet にアクセスする

2. web ブラウザは Actor の Seeder として StoreStream 命令を実行し、メタデータを MainNet に保存し、SubNet を生成し、Navigator と接続する。ストリームのチャンクデータを順次

Trigger から送信する。

ストリームデータの観測

1. web ブラウザは GuestNode にあたるので、Portal ノードのトランスポートアドレスを何らかの方法で入手し、PortalNode と接続し、MainNet にアクセスする

2. web ブラウザは Actor の User として ConnectSubNet 命令を実行し、Navigator を経由して Seeder の SubNet へ接続する。

3. web ブラウザは SubNet の FindStreamMetaTarget 命令を実行し、SubNet 上でメタデータのペイロードに記載されている first チャンクデータの実体データを探索し、入手する。あとは next chunks の探索を続けることでストリームデータの観測を行う

3.6 実装

すべての実装において、使用したプログラム言語は TypeScript である。

3.6.1 ライブラリ

Node.js と web ブラウザ上で動作するように実装を行った。P2P 通信部部には WebRTC を用いている。

3.6.2 ベンチマーカー

通常の Kademlia と本手法の LayeredKad の性能比較を行うために、Node.js 上で動作するベンチマーカーの実装を行った。なお、ベンチマーカーは実装の都合上 P2P 通信部分は WebRTC ではなく、UDP を使用して WebRTC の挙動を再現している。

3.6.3 サンプルプログラム

実際に web ブラウザ上で LayeredKad を用いてライブストリーミングの配信と視聴を行うサンプルプログラムを作成した。Web のフロントエンド部分は React[*2] を使用した。P2P 通信部部にはブラウザ搭載の WebRTC を用いている。

動作確認の結果、ローカル環境下で数ノード (5 ノード以上) の LayeredKad ネットワークを構築した状況ではライブストリーミングの配信/視聴が可能であることを確認できた。

[*2] https://reactjs.org/

第 4 章

評価実験

本論文では、データ通信量と、タスクの処理時間の 2 つの観点から、LayeredKad の評価を行う。現実的なユースケースを想定し、幾つかのファイルがネットワーク上に共有され、それぞれのファイルに関心があるユーザがそれぞれのグループを形成して共有しあっているケースを想定し仮説を立てる。

4.1 データ通信量

4.1.1 仮説

LayeredKad では共有するデータごとにネットワークを分割しており、データの保存や探索がネットワーク全体に影響しないので、Kademlia とノード数が同じであれば、ネットワーク全体における、データ通信量が少なくなることが予想される。ファイル共有サービスにおいてファイル共有速度の最も大きなボトルネックはネットワークの通信速度であるので、データ通信量が少ないことはメリットである。

4.1.2 実験

実験方法

データ通信量の計測を行うために、Kademlia と LayeredKad でデータ通信量の計測を行うベンチマークプログラムを作成した。ベンチマークプログラムは表 4.1 の環境で実行した。

表 4.1 スペック

CPU	Intel Core i7-7500U 2.70GHz 2Core 4Threads
OS	Ubuntu 19.10

本ベンチマークプログラムでは、ノードのペアを 1 グループとして、ファイル共有を行わせる。ベンチマークプログラムのシナリオのイメージを図 4.1 に示す

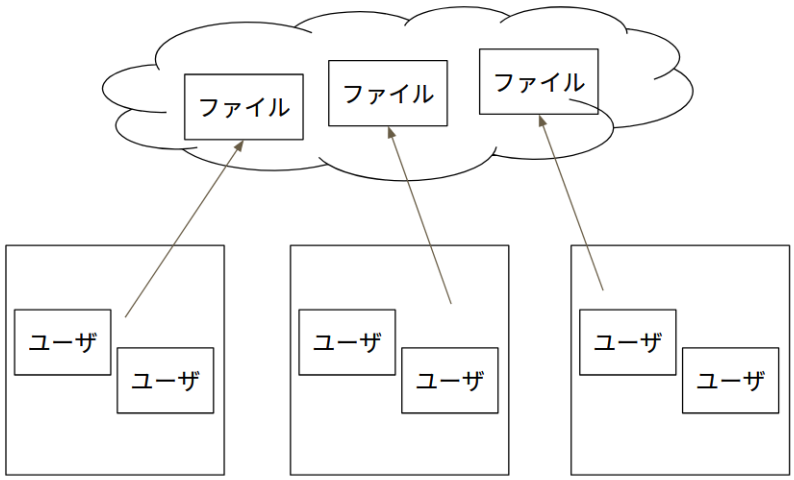

図 4.1　通信量ベンチマークのイメージ

ノード数を変数パラメータとして、パラメータを増加させて LayeredKad と Kademlia のデータ通信回数を記録し、比較を行う。

実験結果

表 4.2　実験結果

ノード数	Layered Kad(回)	Kademlia (回)
10	10	380
30	16	729
40	32	1026
50	33	1278

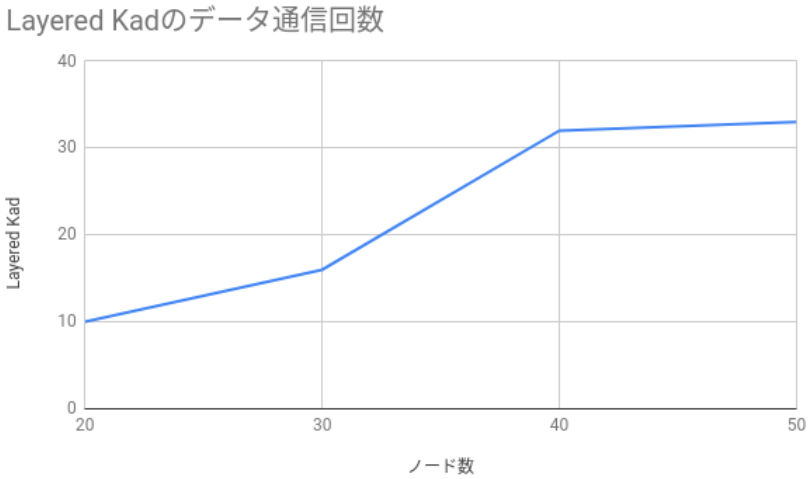

図 4.2　layeredKad の通信量

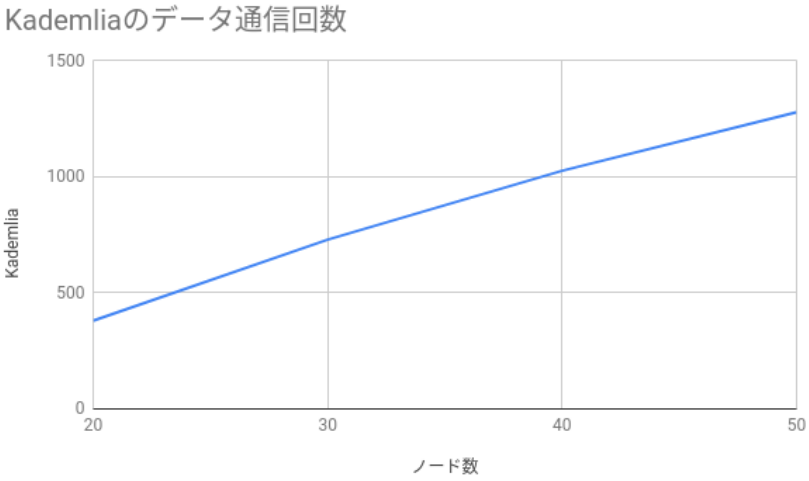

図 4.3　kademlia の通信量

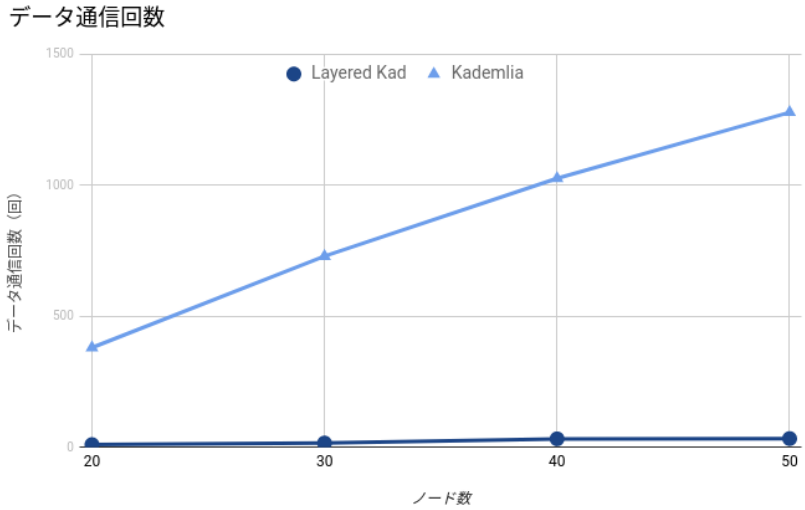

図 4.4　通信量の比較

4.1.3　考察

実験結果は、仮説の通りの結果となった。ただし、この結果はベンチマークのシナリオが LayeredKad が共有するデータ毎に別のネットワークを構築するという性質に有利な設定であるため、これほどの性能差が出たと考えられる。

4.2　タスクの処理時間

4.2.1　仮説

LayeredKad は Kademlia をさらに構造化したシステムであるため、単純に一つのファイルを共有するようなケースでは、当然、Kademlia の方が高速である。しかし、今回のケースでは Kademlia はすべてのファイルについての処理においてもネットワーク全体で処理を行う必要があるのに対して、LayeredKad ではユーザグループ（SubNet）内で完結するため、処理効率がよく、タスクの処理時間も短くなると考えられる。

4.2.2　実験

実験方法

タスク処理時間の計測を行うために、Kademlia と LayeredKad でタスク処理時間の計測を行うベンチマークプログラムを作成した。ベンチマークプログラムは表 4.3 の環境で実行した。

表 4.3　スペック

CPU	Ryzen Threadripper 2950X 3.50Ghz 16Core 32Threads
OS	Ubuntu 18.04

ベンチマークプログラムのシナリオはデータ通信量のベンチマーカーと同じであるが、タスク処理時間をより正確に計測するために、1 ノードに付き、CPU のスレッド 1 つを割り当てるようにベンチマークプログラムのマルチスレッド化を行っている。本実験では、ノード数は 16 ノードで固定とし、共有するデータ数のチャンキング数を変数として実験を行う。

実験結果

表 4.4　実験結果

チャンク数	Layered Kad (s)	Kademlia (s)
1	2.098	0.636
4	2.728	2.032
5	2.469	2.435
6	2.581	2.679

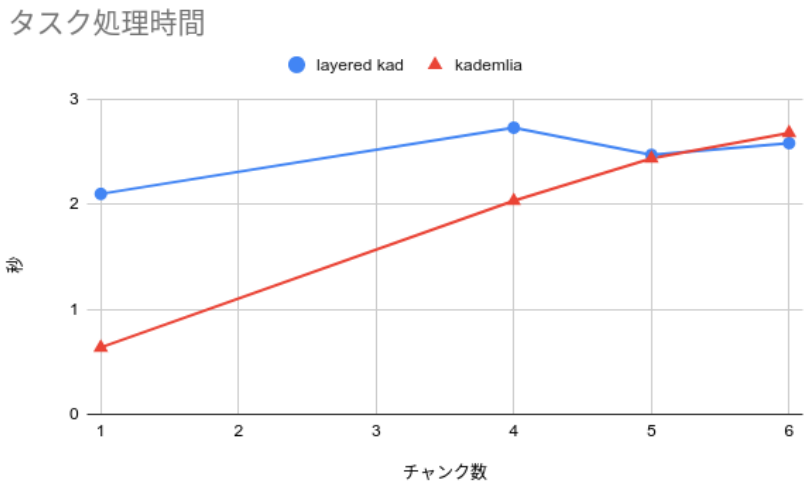

図 4.5　タスク処理時間の比較

4.2.3　考察

LayeredKad ではデータのチャンキング数が増加してもタスク処理時間は概ね変化しないのに対して、Kademlia ではチャンキング数の増加に対してタスク処理時間が比例増大していることがわかる。LayeredKad では、いくらチャンク数が増えたとしても SubNet のペアとなるノードとの一

対一の通信回数がチャンク数分増えるだけなのに対して、Kademlia はチャンク数分、ネットワーク全体に対して問い合わせるため、その分処理時間が増大していると考えられる。

第 5 章

おわりに

5.1 結論

本論文では、Kademlia をベースに効率的に静的なデータやストリーム形式のデータを共有できるシステムの開発を行った。ネットワークを共有するデータ毎に分割し、非効率的なデータ通信を削減するための仕組みである LayeredKad を考案し、実装した。静的なデータや動的なデータなどを柔軟に扱えるようにするために、メタデータを共有する仕組みを考案した。メタデータを共有するネットワークを MainNet とし、メタデータの実体データを共有するネットワークを SubNet とし、ネットワークの分割を実現した。

LayeredKad の性能を評価するため、ユーザが実利用する際のシチュエーションに沿ったシナリオ上で、LayeredKad と Kademlia のベンチマークを行い性能の評価を行った。ベンチマークはネットワークのボトルネックとなるデータ通信と、タスク処理時間の二項目に着目して行った。データ通信のベンチマークでは、LayeredKad はそのネットワーク分割能力によって Kademlia よりデータ通信量が小さいごとを実証できた。タスク処理時間のベンチマークでは、Kademlia がチャンク数の増加に対して処理時間が比例増大したのに対して、LayeredKad ではチャンク数の増加が処理時間に影響しないことがわかった。

5.2 課題

Kademlia はネットワーク全体に一定のルールでデータを保存しているが、LayeredKad では実体のデータはデータを利用するユーザ間でのみデータが保存されている。そのためデータの冗長性という面では Kademlia に劣っている。また、Kademlia は様々なサービスに応用され長期間の安定した動作が確認されているが、本論文では LayeredKad の長期間の安定動作については検証しておらず、未知の問題が存在する可能性がある。

謝辞

本研究を進めるにあたり, ご指導を頂いた指導教員の萩原助教授に感謝致します. 日頃の議論において助言や知識を頂いた萩原研究室の皆様に感謝します.

参考文献

[1] Bittorrent (btt) - the token that will enable blockchain mass adoption. https://www.bittorrent.com/btt/. (2020 年 1 月 16 日 閲覧).

[2] Bittorrent. https://www.bittorrent.com/lang/ja/. (Accessed on 01/16/2020).

[3] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems*, pages 53–65. Springer, 2002.

[4] 高野祐輝, 井上朋哉, 知念賢一, and 篠田陽一. Nat 問題フリーな dht を実現するライブラリ libcage の設計と実装. コンピュータ ソフトウェア, 27(4):4_58–4_76, 2010.

[5] Webrtc home — webrtc. https://webrtc.org/. (Accessed on 01/16/2020).

[6] Jonathan Rosenberg. Interactive connectivity establishment (ice): A protocol for network address translator (nat) traversal for offer/answer protocols. 2010.

[7] Dan Wing, Philip Matthews, Rohan Mahy, and Jonathan Rosenberg. Session traversal utilities for nat (stun). 2008.

[8] Philip Matthews, Rohan Mahy, and Jonathan Rosenberg. Traversal using relays around nat (turn): Relay extensions to session traversal utilities for nat (stun). 2010.

[9] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext transfer protocol–http/1.1. 1999.