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

Examination and Evaluation about 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

Distributed hash tables have been applied to file sharing services, are already widely used, and have recently attracted attention as a technology related to blockchain. Because it is highly resistant to churn, it can be applied to actual services and is used by many services. In recent years, demand for live streaming services has been increasing. On the other hand, the distribution platform is highly dependent on the enterprise, and the service user's position is weak. In this paper, we implement and evaluate the underlying system of a live streaming service based on Kademlia. The simple implementation of live streaming in Kademlia reduces communication efficiency, but in this paper we can reduce inefficient communication by customize Kademlia networks. 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システム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] 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が難しく、 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。ファイル共有サービスが分散型ファイル共有サービスによって, リソースの分散化に成功したようにライブストリーミングサービスもリソースの分散化が行われることが望ましいと考えられる.

1.2 目的

第 1.1 節で述べたように, ライブストリーミングサービスの分散化には意義がある. 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, Kademlia という分散ハッシュテーブルをベースに実装する.

[*1] https://www.youtube.com/live
[*2] https://www.twitch.tv

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．そのため, Kademlia を更に構造化した LayeredKad という手法を提案する．

LayeredKad 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．ライブラリ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．

完成したライブラリを用いて Web ブラウザ上で動作する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動作確認を行う．LayeredKad のライブラリを用いたベンチマークプログラムと 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．本研究のソースコードは Github 上のリポジトリ [6] で GPLv3[*5] ライセンスで公開している。

[*3] ノードの出入りに対する耐性の強さ

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

[*5] https://www.gnu.org/licenses/gpl-3.0.html

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, あるデータとそのデータ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する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.

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多く利用されている.

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 Key

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 Value と対になる Key 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 からなる二分木状の経路表を持っている．一つの k-bucket には K 個 (本研究では 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.

- 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.

- STORE
  対象ノードに Value, Key の組を保持させる. 保持させる際のルールは, Key に対して FindNode を k-buckets の Key に対する XOR 距離が近い上位 K 個のノードが固定されるまで繰り返し実行し k-buckets に Key にできるだけ XOR 距離が近いノードが入るように k-buckets の最適化を行う。次に自身の k-buckets から最も Key に XOR の距離が近いノードを近い順に K 個選択し, そのノードらに Value, Key の組を与える.

- FIND_VALUE
  自身の k-buckets のうち最も Key に XOR 距離の近いノードに Key に対応する Value を持っているか問い合わせる. 問い合わせられたノードは Value を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 XOR 距離が近い上位 K 個のノードの情報を返す.

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のノードに返す. 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 距離が近い上位 K 個のノードに対し自身のノード ID を Key とした FIND_NODE を自分の ID に対する XOR 距離が近い上位 K 個のノードが固定されるまで繰り返す。

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，Windows,Linux,Mac,Android や iOS といったネイティブ環境で動作する libwebrtc[*1] などの実装が公開されている．WebRTC には NAT 越えを実現するために ICE[7] という仕組みを採用している.

WebRTC には任意のデータを通信するための DataChannel と音声や動画などのメディアを通信するための MediaChannel の 2 種類の通信方法が存在する. 本研究では Kademlia の実装における UDP の代換えとして WebRTC を用いるため, 通信方法に DataChannel を利用する.

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[8] 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，インターネット上に STUN サーバが存在する必要がある.

TURN[9] 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

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

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 と ICE Candidate 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，シグナリングのため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.

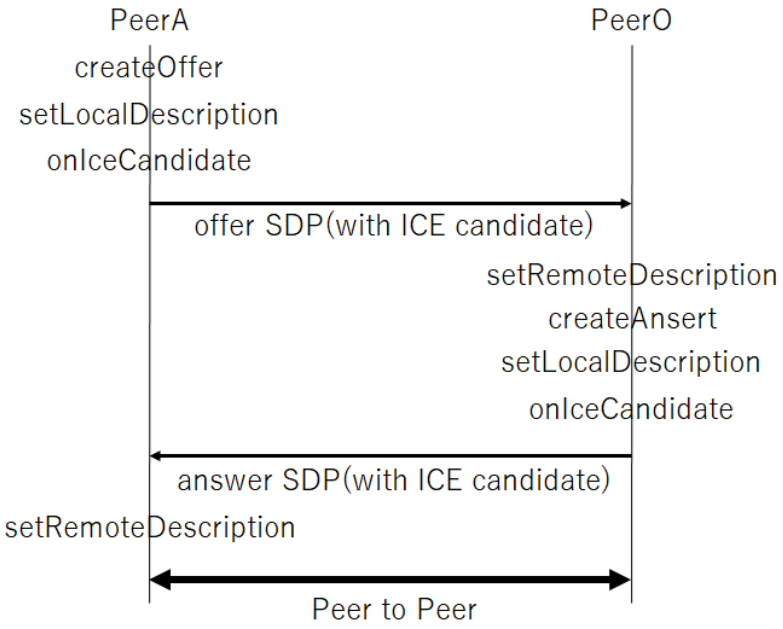

図 2.1　シグナリング

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

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

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 をベースに改良を施し、効率的にストリーム形式のデータを共有できるようにしたシステムである 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, FindNode を繰り返し実行し, k-buckets を最適化する. その際に, ネットワーク上の不特定多数のノードと通信した上、最大で K 個の不特定なノードにデータを保存することになる. 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の不特定多数のノード ("チャンク数 $\times$ K" 個) に保存することになる. 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, 全ノードが参加する 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ノード同士でさらに別の 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.

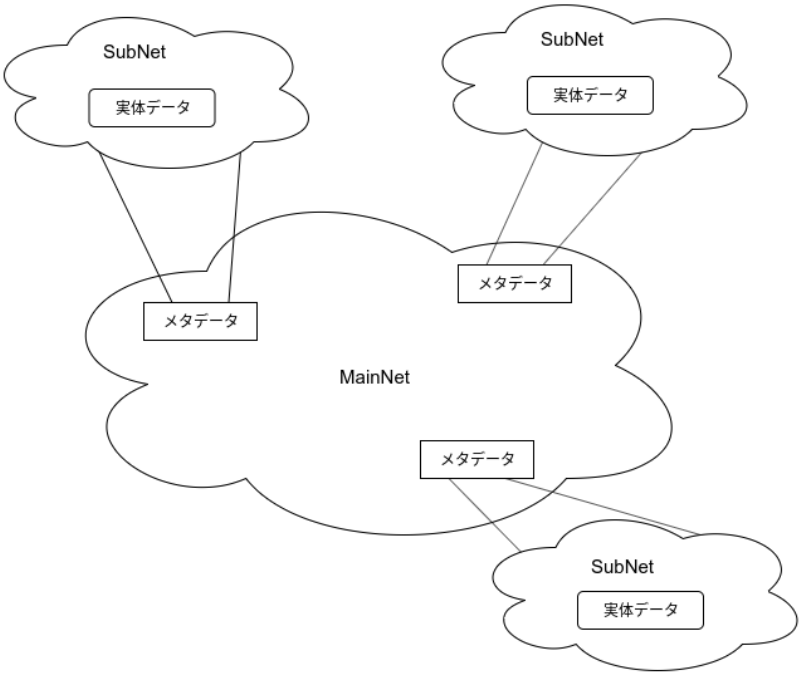

図 3.1　MainNet と SubNet

LayeredKad における幾つかの用語は BitTorrent の RFC[10] の影響を受けている.

3.2　Kademlia の拡張

Kademlia の Store は単に Value と Key(Value のハッシュ値) のペアしか情報を持たない. しかし, ストリームデータを扱う際にチャンクデータ同士の関係性を表すためには Value と Key だけでは情報が不足する. そのため LayeredKad の Kademlia では Value と Key のペアに対し任意の文字列をアノテーションすることを許可する拡張を行う. このアノテーションは現状ストリームデータの共有にのみ用いられる.

拡張された Store の型定義を示す.

```
1         type Store =
2                 (key: string, value: any, msg?: string) => void
```

Key は Value の sha1 ハッシュ値である必要があるが, msg はアノテーションであり, 任意の文字列を取ることができる.

3.3　ストリームデータ

3.3.1　ストリームデータの保存

ストリームデータは動的に生成される複数のチャンクデータから成り立つ．LayeredKad では一つの ID(最初のチャンクのハッシュ値) 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 chunk とする) の次に生成されるチャンクデータ (便宜上 next chunk とする) が生成されるのを待ち, next chunk のハッシュ値を Key-Value の msg アノテーションとする．ストリーミングを終了する際には msg にハッシュ値ではなく、”complete”文字列を与える。チャンクデータを保存する様子を図 3.2 に示す.

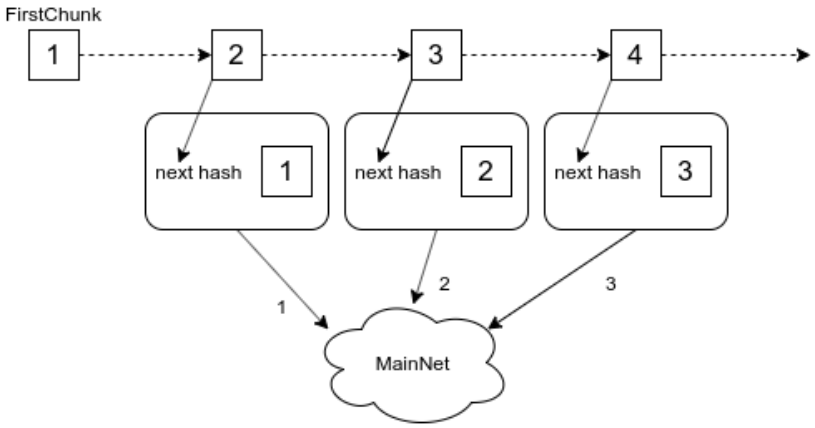

図 3.2　ストリームデータの保存

3.3.2　ストリームデータの探索

図 3.3 のように FirstChunk の Key を元に探索した Value の msg アノテーションを辿ることで, ストリームデータを動的に継続的に取得できるようになっている．msg の中がハッシュ値ではなく”complete”文字列なら、ストリームデータが終了したとみなし、ストリームの観測を終了する。

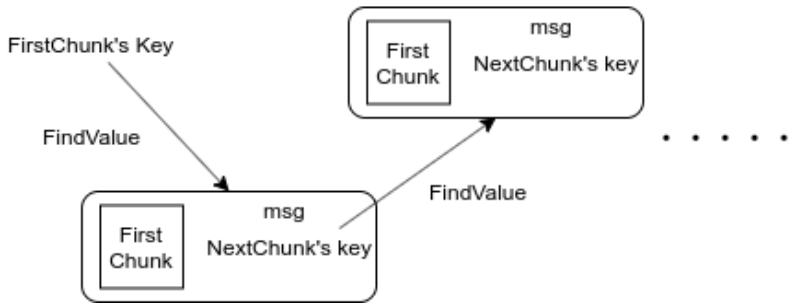

図 3.3　ストリームデータの探索

3.4　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共有するデータの情報メタデータとして共有し, メタデータの情報に従ってノードの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.

本研究ではこの基本的な Meta データ構造を拡張した StaticMeta と StreamMeta が存在する

3.4.1　StaticMeta

静的なデータを扱うメタデータである。

```
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. 実体データを SubNet で探索する際には、keys のそれぞれの Key を FindValue で探索し、最後に結合する。

3.4.2 StreamMeta

ストリーム (ライブ映像など) を扱うメタデータである。

```
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. 実体データを SubNet で探索する際には、payload の first を FirstChunk の Key とし、3.3.2 節のストリームデータの探索方法に従い、ストリームデータをサブスクライブする。

3.5 Network

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

3.5.1 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, メタデータの実体データを扱いたいノードと SubNet との橋渡しを行う. MainNet のインスタンスは一つのノードにつき一つのみ存在する。

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, 公開トランスポートアドレスを持つノードを PortalNode と呼ぶ.

[*1] IP アドレスとポートのペア

ノードの接続方法

LayeredKad の MainNet のノード間では最終的に 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 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 (1 か 2 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全ノードが 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.

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.

3.5.2 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. そのため、SubNet のインスタンスは一つのノードに複数存在しうる。

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, 通信方法によるノードの分類は存在しない.

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, 探索結果を返す.

- 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
```

  SubNet の持つメタデータを元にストリームデータをサブスクライブする. チャンクデータのアノテーションから次に FindValue すべき key を入手している.
  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渡す.

3.6 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 User, Seeder, 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.

3.6.1 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, メタデータを持つ Navigator を仲介して SubNet と接続する. SubNet と接続している User を特に ObserveUser と呼ぶ. また, ObserveUser は Seeder の役割を兼ねる.

User のインスタンスは一つのノードにつき一つのみ存在する。

SubNet への接続

詳細な SubNet への接続手順を示す.

1. メタデータの URL をもとに MainNet でメタデータを探索する

2. メタデータを返したノード (Navigator) に Seeder への接続仲介を要請する
3. Seeder との接続完了は SubNet へ接続できたことを意味する
4. ObserveUser は Seeder のインスタンスを生成し, 接続している SubNet における Seeder を兼任する
5. Seeder は Navigator を兼任するため, User は Navigator のインスタンスも生成する

User の命令

User 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, メタデータを返したノードを 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.

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

  メタデータの url を受け取り, SubNet とメタデータを返す.

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ことができる.

3.6.2 Seeder

Seeder とはデータの配信を行うノードである. 実際のユースケースでいうところのデータのアップロード者などにあたる. Seeder は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 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.

User が SubNet 内でメタデータの実体データを探索する際に Seeder は実体データを所持している場合, そのデータを User に渡す. (なお, このプロセスは Kademlia の FindValue に従う)

Seeder は SubNet 毎に存在するので、Seeder のインスタンスは一つのノードに複数存在しうる。

SubNet の生成

Seeder 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. そして Seeder はメタデータの情報を持った SubNet を生成し peers の対象のノードたちを Navigator にする.

Seeder の命令

Seeder には StoreStatic と StoreStream 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 とメタデータを返す.

- 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, メタデータの実体であるストリームのチャンクデータを SubNet に動的に保存する.

  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.

3.6.3 Navigator

Navigator は User を SubNet に接続する際の仲介を行う 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.

Navigator は自分の所持しているメタデータ毎に存在するので、Navigator のインスタンスは一つのノードに複数存在しうる。Navigator は SubNet には参加していない。

Seeder との接続

LayeredKad のノードは MainNet 上でメタデータを Store されると, Navigator になるので, 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 と表現する. NavigatorCandidate は MainNet を監視しており, NavigatorCandidate 自らにメタデータを Store された時に, Store してきたノード (Seeder) に対してシグナリングを行い WebRTC の peer を生成しコネクションを作る. 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 Navigator 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 Navigator 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 WebRTC の 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.

3.7 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.

3.7.1 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.

3.7.2 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 3.3.2 節のように msg アノテーションを辿りチャンクの探索を続けることでストリームデータの観測を行う。

3.8 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.

3.8.1 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.

3.8.2 ベンチマーカー

通常の Kademlia と 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.

3.8.3 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. 本実験では, 現実的なユースケースとして, 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データ毎にグループを形成し共有しあっているケースを想定し仮説を立て、実験検証を行う. 評価実験のシナリオのイメージを図 4.1 に示す。

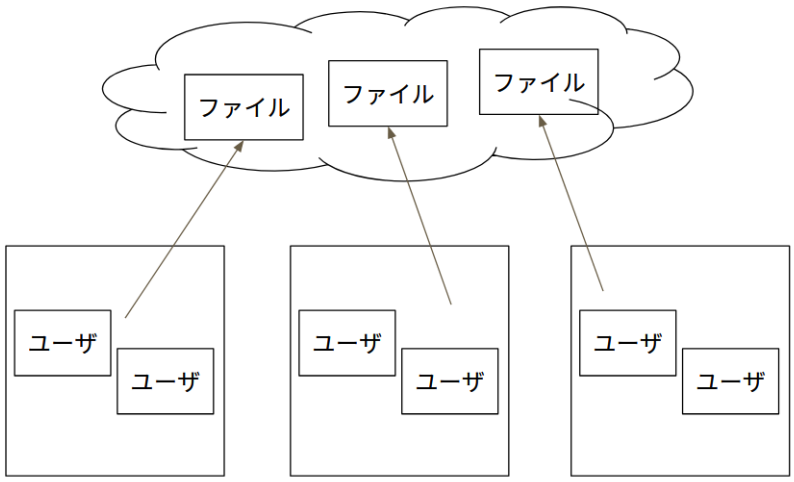

図 4.1　評価実験のシナリオのイメージ

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り, 特にライブストリーミングデー

タは大容量なデータを高頻度にやり取りする必要があるため, データ通信量が少ないことは大きなメリットである.

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. ノード数を変数パラメータとして, パラメータの値を増加させ, LayeredKad と Kademlia のデータ通信回数を記録し, 比較を行う.

実験結果

Kademlia と LayeredKad の実験結果の表を表 4.2 に
Kademlia の実験結果のグラフを図 4.2 に,
LayeredKad の実験結果のグラフを図 4.3 に,
Kademlia と LayeredKad の実験結果のグラフを 4.4 に示す.

表 4.2 Kademlia と LayeredKad の実験結果

ノード数	Kademlia (回)	LayeredKad (回)
10	90	5
20	380	10
30	600	16
40	779	26
50	965	36
60	1132	48

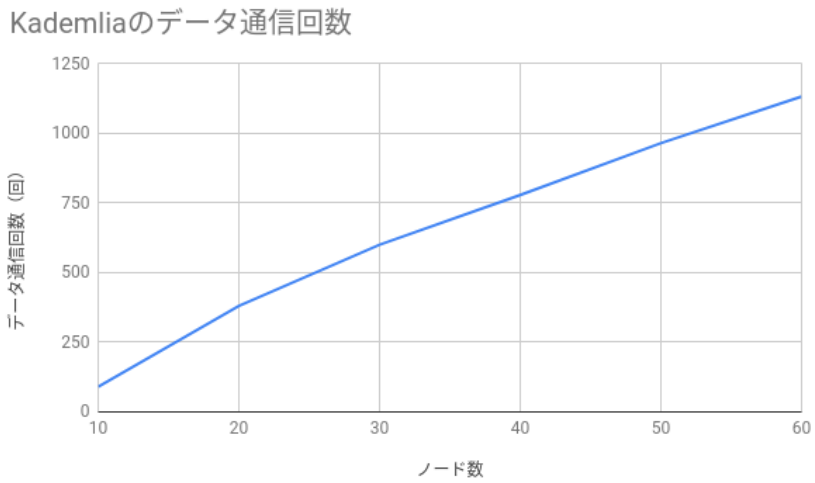

図 4.2　kademlia の通信量

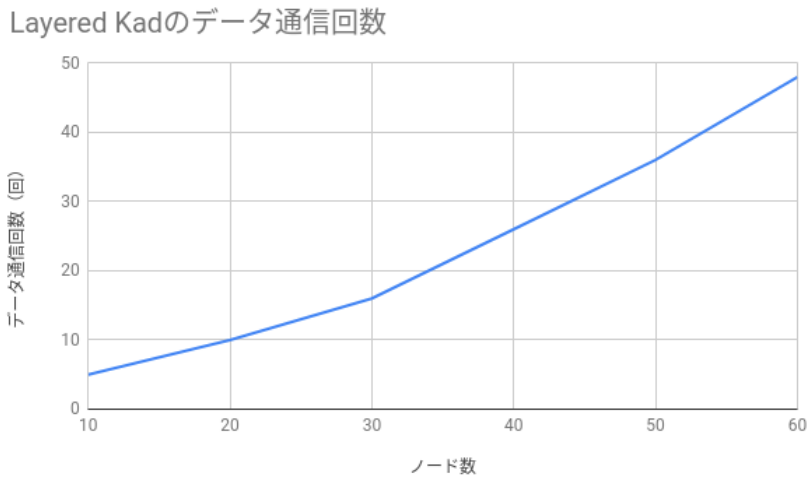

図 4.3　layeredKad の通信量

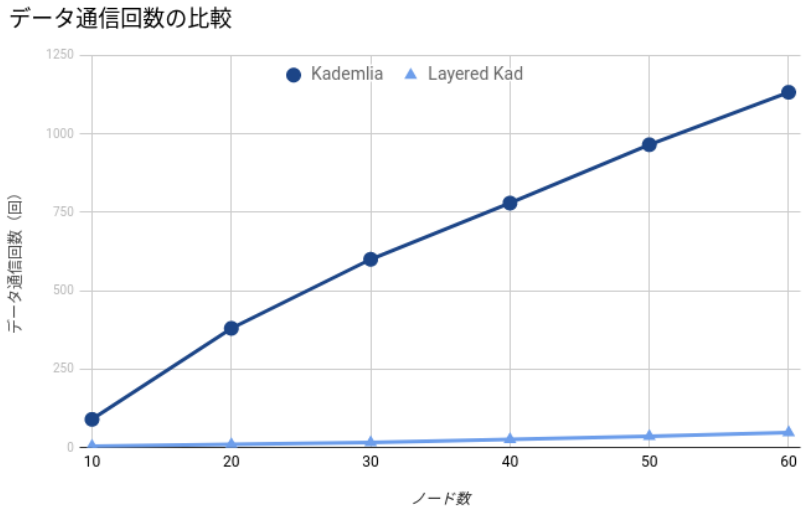

図 4.4　通信量の比較

ノード数がいずれの場合においても, データ通信回数は LayeredKad が Kademlia を大きく下回った.

4.1.3　考察

実験結果は, 仮説の通りの結果となった. Kademlia は一つのデータを保存する際に, k-bucket のサイズ分 (本論文では 20) のノードとデータのやり取りをすることになるため, この実験結果のような通信回数になったと考えられる.

一方 LayeredKad はデータ毎に SubNet を生成し, その内部でデータのやり取りを行うため, 本実験の場合データをノードのペアがやり取りを行うシナリオとなっているため SubNet にはノードが 2 つしか無く, データのやり取りは 1 回で済む. そのため, Kademlia より通信回数が大幅に少なくなったと考えられる.

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, 単純に一つのデータをネットワーク全体で共有するようなケースでは, 処理のステップ数の少ない Kademlia の方が高速であると予想される. しかし今回の実験ケースでは複数のデータを複数のユーザグループ毎に共有しているので, Kademlia はすべてのデータにおける処理をネットワーク全体で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

Kademlia と LayeredKad の実験結果を表 4.4 と図 4.5 に示す.

表 4.4 実験結果

チャンク数	Kademlia (s)	Layered Kad (s)
1	0.636	2.098
4	2.032	2.728
5	2.435	2.469
6	7.679(2.679)	2.581

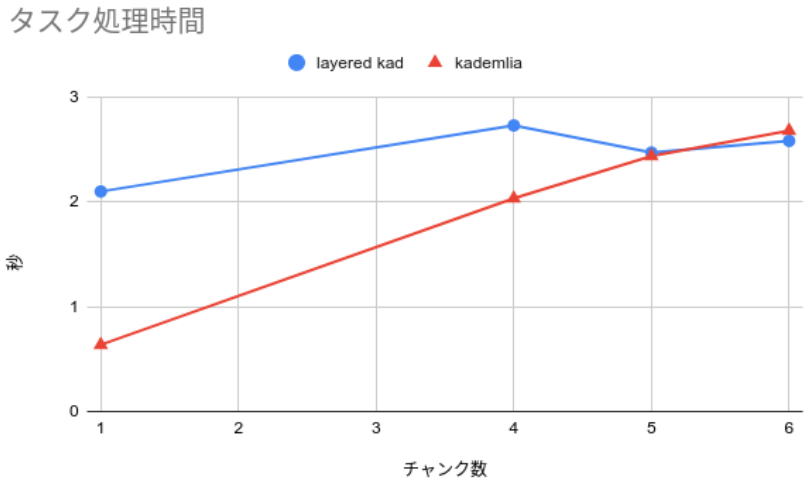

図 4.5　タスク処理時間の比較

表 4.4 のチャンク数 6 での Kademlia の結果を”7.679(2.679)”としているのは, Kademlia がこのベンチマークでタイムアウトを引き起こしていたので, タイムアウトの時間 5 秒を引いた値を参考値として括弧の中に表記している. また, 図 4.5 のグラフの値は括弧中のものを使用している.

図 4.5 のグラフより Kademlia はチャンク数の増加に比例して, タスク処理時間が増大していて LayeredKad はチャンク数の増加が, タスク処理時間にあまり影響していないことがわかる.

4.2.3　考察

Kademlia で発生したタイムアウトの原因

本ベンチマークでは, 複数のノードが全く同時にデータの共有タスクを実行しているので, 現実のユースケースより, むしろ高負荷であるため, 内部でデッドロックが発生してタイムアウトが生じたと考えられる. 本研究の LayeredKad の MainNet と SubNet 内部の Kademlia 実装部分は, 比較用の Kademlia 実装とソースコードが同一であるため, 同一条件下で LayeredKad ではタイムアウトが生じていないという点も LayeredKad の優位性を示せていると考えられる.

LayeredKad と Kademlia の比較

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原因は 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からだと考えられる.

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. メタデータによる柔軟なデータ定義, 現実的な特定のユースケースにおける高パフォーマンスな性能によって, LayeredKad が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基盤としての有用であることを示せたと考えられる.

5.2 課題

本研究では LayeredKad の SubNet のノード数が少ない条件でしか実験が出来ていない. そのため, SubNet のノード数が増大 (K より大幅に多い数に) した際には通常の Kademlia と同じような問題が発生すると考えられる. (SubNet 内部のネットワーク実装は通常の Kademlia と同一であるため.) この問題を回避する案として, さらに SubNet を分割し構造化するなどといったアイデアが考えられるが, 本研究では実装の範囲外とした.

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/. (Accessed on 01/16/2020).

[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] shinyoshiaki/layered-kademlia. https://github.com/shinyoshiaki/layered-kademlia. (Accessed on 01/30/2020).

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

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

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

[10] bep_0000.rst_post. http://www.bittorrent.org/beps/bep_0000.html. (Accessed on 01/25/2020).