Vol.2017-IOT-39 No.9
Vol.2017-SPT-25 No.9
2017/9/29

WebRTCを用いた耐故障性の高いウェブブラウザ間構造化P2Pネットワークの実現

鄭焱祖[1]　川井悠司[1]　李佼柯[1]　安倍広多[1]

概要：Webブラウザ同士を直接P2P接続するWebRTC技術を用いたブラウザ間構造化P2Pネットワークの実現手法について述べる．WebRTCには，相手ノードを直接指定してコネクションを確立できない，コネクションの確立にはシグナリングが必要，といった特性がある．本研究では，これらの特性を考慮したブラウザ間コネクション確立方式を提案する．提案するブラウザ間コネクション確立手法はシグナリングに特定のサーバを用いないため，耐故障性とスケーラビリティに優れている．提案手法はコネクション管理ライブラリWebRTC-Managerとして実装した．さらにWebRTC-Managerを用いて，Web環境で動作する双方向リング構築アルゴリズム $\mathrm{DDLL_{web}}$ とキー順序保存型構造化P2PネットワークKirinを実現した．本稿ではこれらの詳細について述べる．

キーワード：WebRTC，構造化P2Pネットワーク，ウェブブラウザ，耐故障性

An Implementation of a Fault Tolerant Web Browser-Based Structured P2P Network over WebRTC

Zheng Yanzu[1]　Yuji Kawai[1]　Jiaoke Li[1]　Kota Abe[1]

Abstract: In this paper, we describe a method to implement Web browser-based structured P2P network using WebRTC technology, which establishes a connection between 2 browsers directly. WebRTC has the following characteristics; to establish a connection, 1) the target node cannot be specified directly and 2) *signaling procedure* is required. We propose a method to establish a WebRTC connection between 2 browsers that takes these characteristics into account. This method has good fault tolerance and scalability because it does not use any server for signaling. The proposed method is implemented as a connection management library *WebRTC-Manager*. Furthermore, using WebRTC-Manager, we have implemented a doubly linked ring managing algorithm $DDLL_{web}$ and key-order preserving structured P2P network *Kirin*, both of which run in Web environment. This article describes these details.

Keywords: WebRTC, Structured P2P Network, Web Browser, Fault Tolerance

1. はじめに

2つのウェブブラウザ間を接続･通信するためのWebRTC規格が主要なブラウザで利用できるようになりつつあることに伴い，WebRTCを用いてP2Pネットワークを構築する，**ブラウザ間P2Pネットワーク**がいくつか実装されている（[1][2][3]など）．従来のP2Pネットワークを利用したサービス（Skype，BitTorrentなど）では，ユーザは利用するために専用のアプリケーションをインストールする必要があるが，ブラウザ間P2Pネットワークを利用したサービスはウェブブラウザさえあれば利用できるため，ユーザにとって利便性が高い．

WebRTCには以下の特徴がある．WebRTCはブラウザ間にコネクションベースの通信路を提供するが，(1) 直接相手ノードを指定してコネクションを確立することはできない（相手を指定するためのIPアドレスのようなロケー

[1] 大阪市立大学大学院創造都市研究科
Graduate School for Creative Cities, Osaka City University

タは存在しない）．(2) コネクションを確立するにはコネクション確立前に，別のノードを介して，相手ノードとの間でコネクション確立に必要な情報を交換する必要がある（シグナリング手続き）．

従来の P2P ネットワークは，基本的に相手ノードのロケータを知っていればいつでも直接通信できることを前提として設計されているが，WebRTC ではこの前提が成り立たない．このため，ブラウザ間 P2P ネットワークでは，既存の P2P ネットワークをそのまま用いることはできず，WebRTC の特徴を考慮した設計を行う必要がある．

本研究では，まず WebRTC の特性を考慮したブラウザ間 P2P ネットワークを実現する方式を検討した．既存のブラウザ間 P2P ネットワークでは，シグナリングのために中央サーバを用いるものが多いが，この方式は耐故障性とスケーラビリティ上の問題がある．提案方式ではシグナリングに P2P ネットワークを用いる．ただし，P2P ネットワークに新規に参加するノードのために Node.js[*1] 上で動作するノード（**ポータルノード**）を導入する（4.1.1 節で詳述）．提案方式は **WebRTC-Manager** と呼ぶライブラリとして実装した．WebRTC-Manager は，提案方式に基づいてコネクション確立・切断などを行う API を提供する．

さらに本研究では，WebRTC-Manager が提供する API を用いてブラウザ間構造化 P2P ネットワーク（**Kirin**）を実現した．Kirin は著者らが提案している（非ブラウザ用の）構造化 P2P ネットワーク Suzaku[4] をベースとしている．Suzaku（およびそれに基づく Kirin）はキー順序保存型構造化 P2P ネットワークであり，キーが隣接するノードは P2P ネットワーク上でも隣接する性質を持つため，キーの範囲を指定した範囲検索やアプリケーションレベルマルチキャストを実現できる．このため，さまざまなアプリケーションに有用と考えられる．

Kirin はノードがキーの昇順でソートされた双方向リングを必要とする．このため，同じく著者らが提案している P2P ネットワークに適した分散双方向リング構築アルゴリズム DDLL[5] を，WebRTC-Manager を用いてブラウザ間 P2P ネットワークで動作するように修正した（$\mathbf{DDLL_{web}}$ と呼ぶ）．

以下，2 章で WebRTC について簡単に紹介し，3 章では関連研究について述べる．4 章で提案方式，5 章で実装について述べる．6 章で簡単な評価を与える．最後に 7 章で結論と将来の課題について述べる．

2. WebRTC

WebRTC でノード a とノード b の間でコネクションを確立する手順を述べる．

コネクション確立のために，a と b の間でシグナリングを行うための通信路（シグナリングチャネル）が必要である．一般的には Web サーバ（WebSocket サーバ）が用いられるが，実際には任意の通信路を用いることができる．

a は WebRTC の `RTCPeerConnection.createOffer()` API を用いて，Offer SDP (Session Description Protocol) を生成し，（シグナリングチャネルを通じて）b に送信する．b は `RTCPeerConnection.createAnswer()` API を用いて Answer SDP を生成し，a に送信する．

両ノードでは自ノードの通信端点の候補（ICE Candidate）を収集し，相手ノードに渡す．候補には，ローカルインタフェースの IP アドレスや，STUN サーバから取得した NAT の外側 IP アドレスなどが含まれる．両ノードで ICE Candidate を交換することによって互いに接続可能な通信路を発見し，コネクションが確立する．

ICE Candidate は，見つかるたびに渡す方式（Trickle ICE）と，すべての候補を収集してから渡す方式（Vanilla ICE）がある．すべての ICE Candidate の取得には時間がかかるため，Trickle ICE を用いたほうがコネクション確立までの時間は短いことが多い．

WebRTC では，NAT の内側のノード同士がコネクションを確立できるように，UDP hole punch を行う．最終的にコネクションが確立できない場合は，TURN サーバによるメッセージ中継にフォールバックする．

3. 関連研究

1 章で述べたように，WebRTC のコネクション確立は，直接相手ノードを指定できない，シグナリングトラフィックを中継する必要があるなど煩雑である．このため，これを簡略化するライブラリが開発・利用されている．代表的なライブラリとして PeerJS[6] がある．

PeerJS はピア間でのシグナリングや接続先ノードを指定するために **PeerServer** と呼ばれる中央サーバを用いる．PeerJS を利用するノードは WebSocket で常に PeerServer と接続している．PeerServer は WebSocket コネクションに一意の ID を割り当てる．ノードが WebRTC コネクションを確立する場合は相手ノードに割り当てられた ID を指定し，PeerServer がシグナリングトラフィックを中継する．

この方式は，PeerServer が単一故障点となるため耐故障性が低く，また，シグナリングトラフィックを PeerServer が中継するためスケーラビリティも低い．

PeerJS を利用して構築されたブラウザ間 P2P ネットワークに webrtc-chord[1]，webrtc-kademlia[2] などがある．

4. 提案手法

4.1 方針

ここでは実現するブラウザ間構造化 P2P ネットワークの方針を述べる．

[*1] `https://nodejs.org`

4.1.1 ノード間コネクション確立方式

WebRTC ではコネクション確立のためにシグナリングを行う必要がある．PeerJS のようにシグナリングのために中央サーバを用いると耐故障性およびスケーラビリティ上問題があるため，提案方式では中央サーバは利用せず，代わりに P2P ネットワーク自身でシグナリングを行う．

また，通常の P2P ネットワークで，あるノード p が別のノード q と直接通信する（コネクションを確立する）ためには，p は q のロケータを知っているノードから q のロケータを入手すればよい．しかし，ブラウザ間 P2P ネットワークではロケータが使えないため，別の方法を用いる．

P2P ネットワークにおいて p が q とコネクションを確立するのは，一般的に p が何らかの方法で q を探索した結果である．そこで，提案方式ではブラウザ間 P2P ネットワーク上で，p が q に到達するまでの経路をシグナリングチャネルとして利用する．すなわち，p がある条件を満たすノードとコネクションを確立する際，探索メッセージが p, a, b, q という順序で q まで流れたとすると（ここでは p–a，a–b，b–q 間のコネクションは確立済），パス p–a–b–q をシグナリングチャネルとして利用する．これにより，ロケータがなくても p は任意のノードとコネクションの確立が可能となる．

ただし，この方法はブラウザ間 P2P ネットワークに新規に参加しようとするノードでは利用できない（新規参加ノードは既に P2P ネットワークに参加している既存ノードとの接続がないため）．新規ノードは何らかの方法で既存ノードとコネクションを確立する必要があるが，シグナリングが必須である WebRTC は利用できないため，提案方式では WebSocket を用いる．WebSocket は相手の URL を知っていればコネクションの確立が可能で，シグナリングを必要としない．

しかし，一般的にブラウザは WebSocket のクライアント側にはなれてもサーバ側にはなれないため，WebSocket サーバとして Node.js を用いることにした．Node.js は JavaScript を実行するソフトウェアであり，移植性を考慮すればブラウザと同一の JavaScript コードを実行できる．提案方式では，P2P ネットワークに Node.js 上で動作する**ポータルノード**を参加させる．ポータルノードは P2P ネットワークのノードとしても，WebSocket サーバとしても動作する．新規参加ノードは任意のポータルノードの URL を指定して WebSocket コネクションを確立する．これによって既存ノードとの間で接続ができるため，後は上で述べた既存ノードと同じ方法で任意のノードとコネクションを確立できる．ポータルノードは P2P ネットワーク内に 1 つ以上必要であり，必然的に最初のノードはポータルノードである必要がある．ポータルノードは新規ノードの参加時に使われるだけであり，故障や離脱しても（ほかにポータルノードが残っている限り）問題はなく，単一故障点にはならない．

4.1.2 ブラウザ間構造化 P2P ネットワーク

1 章で述べたように，本研究で実現するブラウザ間構造化 P2P ネットワーク Kirin は，キー順序保存型構造化 P2P ネットワーク Suzaku[4] をベースとする．Suzaku は (1)Churn 時でも最大検索ホップ数が $\log_2 n$ 程度に収まる（n はノード数），(2) キーが大小どちらの方向でも近傍ノードの検索は高速に行える，(3) 構造は単純で実装が容易，といった特徴を備える．Suzaku は Chord$^{\#}$ などの既存のキー順序保存型構造化 P2P ネットワークに比べて，検索性能に大きな影響を与えずに経路表の更新頻度を下げられるため（[4] にて予測），コネクション確立のコストが高いブラウザ間 P2P ネットワークに適していると考えられる．

Suzaku はノードがキーの順序でソートされた双方向リングを必要とする．本研究では Kirin のために双方向リング構築アルゴリズム DDLL[5] を Web 環境向けに修正する（DDLL$_{\mathrm{web}}$）．DDLL はネットワーク上に分散した複数のノードが双方向リングを構成するためのアルゴリズムであり，動的なノード挿入および削除が可能である．DDLL は，(1) 右リンク（キーが次に大きいノード（ただしキーが最大のノードではキーが最小のノード）へのポインタ）は常に正しいノードを指す，(2) 左リンク（キーが次に小さいノード（ただしキーが最小のノードではキーが最大のノード）へのポインタ）は 1 メッセージ伝送時間の遅延で正しいノードを指す，(3) 分散排他制御を用いることなくリング構造の一貫性を保つことが可能，といった特徴を持つ．

4.1.3 ソフトウェアの構造

本研究で実現するウェブブラウザ間構造化 P2P ネットワークは以下の 3 つのレイヤ（u 実体は JavaScript のライブラリ）によって構成する．

WebRTC-Manager ノード間のコネクションを確立・管理するレイヤ

DDLL$_{\mathrm{web}}$ ノード間で双方向連結リストを実現するレイヤ

Kirin DDLL$_{\mathrm{web}}$ 上に構造化 P2P ネットワークを実現するレイヤ

以下それぞれの詳細を述べる．

4.2 WebRTC-Manager

WebRTC-Manager は 4.1.1 節で述べた方式に基づいてブラウザ間 P2P ネットワークにおけるコネクション管理を行う．

WebRTC-Manager では，ブラウザノードとポータルノードが混在した環境で動作する．ブラウザノード間でコネクションを確立する場合は WebRTC を用いるが，片方もしくは双方がポータルノードの場合は WebSocket を用いて

コネクションを確立する（WebRTC に比べて WebSocket のコネクション確立コストは低いため）．

4.2.1 コネクション確立の流れ

WebRTC-Manager を用いて新規参加ノード p がノード q とコネクションを確立するまでの流れを述べる（p が新規参加ノードでない場合は (2) から）．

(1) p は WebRTC-Manager が提供する `connectPortal` API を用いて既知のポータルノードに WebSocket コネクションを確立する．

(2) 次に p は `connect` API を実行する．これにより，**コネクション確立要求メッセージ**が生成される．メッセージには，`connect` を実行したノード p のキー（ID），種別（ブラウザノード/ポータルノード）と，ポータルノードの場合は p の WebSocket サーバの URL が含まれる．

(3) コネクション確立要求メッセージは p の **Forwarder** に渡される．Forwarder はコネクション確立要求メッセージを接続先ノードまで P2P ネットワーク上でルーティングするための仕組みである（詳しくは後述）．

(4) 各ノードの Forwarder がコネクション確立要求メッセージを転送し，q に到達する．

(5) q 上の Forwarder は，自ノードがコネクション接続先ということを判定し，`accept` API を実行する．ここで p と q の種別（ブラウザノード/ポータルノード）が確定するため，確立するコネクションのタイプ（WebRTC/WebSocket）も確定する．

(a)（Case 1）p がポータルノードの場合，q から p に WebSocket コネクションを確立し，そのコネクションを利用して**コネクション確立応答メッセージ**を p へ送信する．メッセージには q のキーが含まれる．

(b)（Case 2）q がポータルノードの場合，コネクション確立応答メッセージを q から p へ，反対の経路を使って送信する．メッセージには，q のキーと q の WebSocket サーバの URL が含まれる．

(c)（Case 3）その他の場合，WebRTC コネクションを確立するための Offer SDP を生成し，反対の経路を使ってコネクション確立応答メッセージを q から p へ送信する．メッセージには q のキーと Offer SDP が含まれる．

(6) p がコネクション確立応答メッセージを受信した場合，

(a)（Case 1 の場合）終了．

(b)（Case 2 の場合）q に WebSocket コネクションを確立し，さらに q の側で接続された WebSocket コネクションと，実行した `accept` とを結びつけるための**コネクション識別メッセージ**を q に送信する．

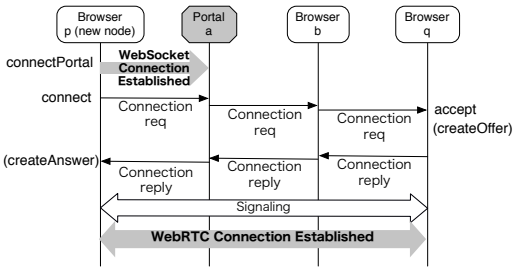

図 1 新規ノードのコネクション確立シーケンス（WebRTC）

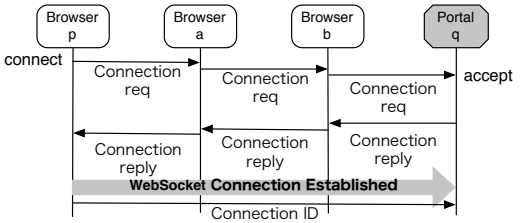

図 2 既存ノードのコネクション確立シーケンス（WebSocket）

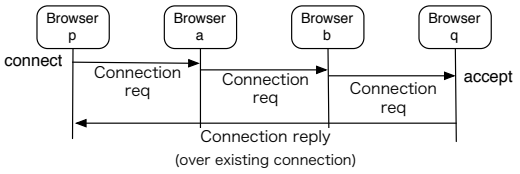

図 3 既存ノードのコネクション確立シーケンス（多重化）

(c)（Case 3 の場合）Answer SDP を作成し，同じパスを使って q に送信する．その後，さらに同じパスを使って p–q 間で ICE Candidate の交換を行い（Trickle ICE），WebRTC コネクションを確立する．

コネクション確立シーケンスの途中で関連するノードが離脱（もしくは故障）した場合，コネクションの確立は失敗する．`connect` と `accept` ではタイムアウトにより呼び出し側にエラーが通知される．

コネクション確立シーケンスの例を**図 1**，**図 2**，**図 3** に示す．図 1 は新規ブラウザノード p がポータルノード a を経由してブラウザノード q とコネクションを確立するシーケンス，図 2 は既存ブラウザノード p がポータルノード q とコネクションを確立するシーケンス，図 3 は既存ブラウザノード p と既存ブラウザノード q の間ですでにコネクションが存在するときに，コネクションを多重化する場合のシーケンスである（多重化に関しては後述）．

4.2.2 Forwarder 機構

提案手法ではコネクションを確立する際のシグナリングを P2P ネットワークによって行うが，P2P ネットワークを構築するのは WebRTC-Manager ではなく上位レイヤの P2P ネットワークレイヤであるため，コネクション確立要求のルーティングは上位レイヤに依頼する必要がある．そのための機構が Forwarder である．

Forwarder の実体は，P2P ネットワークレイヤで用意するコールバック関数であり，すべてのノードで初期化時に

WebRTC-Manager に登録しておく．connect によって生成されたコネクション確立要求は，Forwarder に渡される．Forwarder は次の動作の 1 つを行う．(1) コネクション確立要求を他のノードに転送する，(2) accept を実行し，コネクション確立先となる，(3) reject（後述）を実行し，コネクション確立を拒否する．

P2P ネットワークではさまざまな基準に基づいてノードとコネクションを確立する．このため，動作が異なる複数の Forwarder を使い分けられるようにした．複数の Forwarder を登録しておき，使用する Forwarder は connect の引数で指定する．適切な Forwarder を用いることで，非構造化 P2P ネットワークを含めさまざまなトポロジを実現できる．

4.2.3 コネクションの抽象化と多重化

WebRTC-Manager では，ノードのタイプによって WebRTC コネクションあるいは WebSocket コネクションを確立するが，いずれのコネクションが確立された場合でも上位レイヤに対して統一した扱いを提供するため，WebRTC-Manager の上位レイヤはすべてのコネクションは **PeerConnection** という抽象化したインタフェース（オブジェクト）で扱う．

また，ノード p が connect を，q が accept を実行したとき，既に p–q 間に別のコネクションが確立している場合があるが，同一ノード間に複数の WebRTC/WebSocket コネクションを確立することを避けるため，PeerConnection ではコネクションの多重化も行う．q が accept 実行時に，すでに p との間で WebRTC/WebSocket コネクションが確立している場合は，そのコネクション上に PeerConnection を確立する．

4.2.4 API

WebRTC-Manager の主な API を**表 1** と**表 2** に示す．

connect では，引数として使用する Forwarder とヒントの配列を与える．ヒントは任意のオブジェクトでコネクション確立要求に載せて転送され，Forwarder がコネクション確立要求の扱い（accept するか，reject するか，転送するか）を決定するために用いる．配列となっているのは，複数のコネクションの確立を同時に要求できるようにするためである（DDLL_{web} でノードを挿入するときに利用している）．connect の返り値は PeerConnection を返す Promise（JavaScript で非同期に値を返すために使用するインタフェース）の配列である．

reject は，コネクション確立要求を拒否するために用いる．ブラウザ間 P2P ネットワークでは，コネクション確立の必要性を connect 実行ノードで判断できない場合がある．この場合，とりあえず connect を実行し，Forwarder がコネクション確立が必要ないと判断した場合は reject を実行することでコネクション確立を失敗させる．このとき，コネクション確立応答メッセージによって connect 側に reject で与えた拒否理由が通知される．

表 1 WebRTCManager クラスの主な API

registerForwarder(*forwarder*): void	Forwarder を登録する
onMessage(*handler*): void	メッセージ受信ハンドラを登録する
connectPortal(*url*): Promise<PeerConnection>	ポータルノードとコネクションを確立する
connect(*forwarder*, *hint*[]): Promise<PeerConnection>[]	コネクションの確立を要求する
accept(*connreq*): Promise<PeerConnection>	コネクション確立要求を受け入れる
reject(*connreq*, *reason*): void	コネクション確立要求を拒否する

表 2 PeerConnection クラスの主な API

onDisconnect(*handler*): void	コネクション切断ハンドラを登録する
send(*msg*): void	メッセージを送信する
close(): void	コネクションをクローズする
isConnected(): boolean	コネクションが確立しているかを判定する
getRemoteKey(): any	相手ノードのキーを返す

4.3 DDLL_{web}

4.1.2 節で述べたように，DDLL_{web} は分散双方向リング構築アルゴリズム DDLL[5] を修正したものである．以下，主な修正点について述べる．

4.3.1 データ構造

DDLL では，2 つの変数 (l, r) が左ノードと右ノードのロケータを保持する．しかし DDLL_{web} ではロケータが使えないため，左右のノードは PeerConnection へのポインタで表現する．左あるいは右のノードを x に変更するときは，予め x とコネクションを確立し，当該 PeerConnection インスタンスへのポインタを l もしくは r に代入する．

4.3.2 ノード挿入

DDLL におけるノード u の挿入の流れは概ね次のとおりである（**図 4**）．(1)u はリングに参加済の既知のノード s から u の左右で最も近いノード p と q のロケータを入手する．(2)u は左右のポインタをそれぞれ p と q に設定し，p に SetR メッセージを送信する．(3)SetR メッセージを受信した p は右リンクを u に変更し，SetRAck メッセージを u に，SetL メッセージを q に，それぞれ送信する．(4)SetL メッセージを受信した q は左リンクを u に変更する．（ここでは他ノードの挿入・削除との競合のチェックや，左右のリンクの一貫性確保のための処理は省略している．）

DDLL_{web} ではロケータが使えないため，以下のように左右のノードの検索とコネクションの確立を同時に実行す

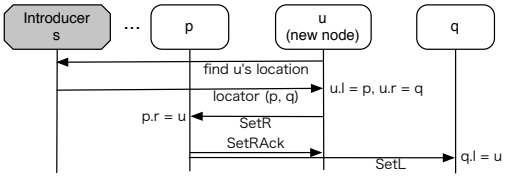

図 4　DDLL のノード挿入シーケンス

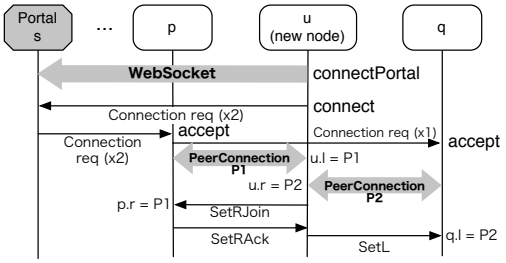

図 5　$\mathrm{DDLL_{web}}$ のノード挿入シーケンス（コネクション確立応答は省略）

る（図 **5**）.

(1) u は `connectPortal` により既知のポータルノード s と接続する.

(2) 次に u は `connect` を実行し 2 本のコネクション確立を要求する．このとき，Forwarder へのヒントとして u のキーを渡す.

(3) 左リンク確立用 Forwarder はヒントを参照し，s を経由して P2P ネットワーク上で u の左ノードとなる p まで 2 本分のコネクション確立要求を転送する．（このとき，後述する Kirin の経路表を用いることでノード数 n に対して $O(\log n)$ ホップで転送する）

(4) Forwarder は，p で 1 本分に関して `accept` する．残りの 1 本分に関しては p の右ノード（q）に転送する（右リンク確立用 Forwarder に切り替える）.

(5) 右リンク確立用 Forwarder は q で 1 本分のコネクションを `accept` する.

(6) u は 2 本のコネクションが確立すると，u の左右のノード（l, r）をそれぞれの PeerConnection に設定し，次に p に SetRJoin メッセージを送信する.

(7) SetRJoin メッセージを受信した p は，右ノード（r）を当該メッセージを受信した PeerConnection に変更し，またその PeerConnection を用いて SetRAck メッセージを u に送る.

(8) SetRAck メッセージを受信した u は SetL メッセージを q に送信する.

(9) SetL メッセージを受信した q は左ノード（l）を当該メッセージを受信した PeerConnection に変更する.

4.3.3　ノード削除

ノード削除のシーケンスを図 **6** に示す（紙数の関係で説明は省略する）.

4.3.4　リングの修復

ノードの障害（削除手続きを実行せずに離脱する場合を含む）が発生した場合，リンクを修復する必要がある．DDLL では，各ノード u は左ノードの障害を検出する．検出した場合は (1)u が保持している近隣ノード集合（十分な数の左方向で近いノードへのポインタ）を用いて左方向で u に最も近い生存ノード x を発見し，(2)u の左リンクを x に変更，(3)x に SetR メッセージを送信して x の右リンクを u に変更，という流れで修復する．しかし，$\mathrm{DDLL_{web}}$ で近隣ノード集合を実現しようとすると，左方向で近い複数のノードに対して常にコネクションを確立しておく必要がありコストが高い．このため，$\mathrm{DDLL_{web}}$ では以下の方法を用いる.

u が左ノード p の故障を検出した場合，修復のために左方向で最も近い生存ノードへコネクションを確立する必要がある．ここで，近隣ノード集合を用いる代わりに，上位レイヤ（Kirin）の経路表を利用する（Kirin が持つ左方向に 2 の累乗個離れたノードへのコネクションを活用する）．まず，u は `connect` を実行する．修復用 Forwarder は上位レイヤの経路表を用いて，u の左側で最も近い，生存しているノード x に効率よく要求を転送し，x が `accept` を実行する（そのようなノードが存在しなければ u 自身が `accept` する）．u と x との間でコネクションが確立したら，u は左リンクを当該 PeerConnection に設定し，また x に SetRJoin メッセージを送信して x の右リンクを当該 PeerConnection に設定する.

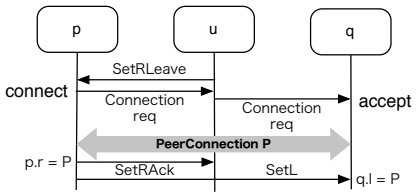

図 6　$\mathrm{DDLL_{web}}$ のノード削除シーケンス（コネクション確立応答は省略）

4.4　Kirin

本節では実現したブラウザ間構造化 P2P ネットワーク Kirin について述べる.

4.4.1　Suzaku

まず，Kirin がベースとする Suzaku の概要を述べる．Suzaku はリングベースの P2P ネットワークで，経路表として Forward Finger Table (FFT) と Backward Finger Table (BFT) を持つ．どちらもノードへのポインタの配列であり，FFT[0] は右ノード，BFT[0] は左ノードへのポインタを保持する．レベル 1 以上は次の方法で更新する[*2].

ノード u が FFT[i] ($i > 0$) を更新する場合（BFT の場合は FFT と BFT を読み替える），u.FFT[$i-1$] が指すノード x から，x.FFT[$i-1$] に格納されたポインタを入手し，u.FFT[i] に代入する（アクティブ更新）．また，このとき

[*2] Suzaku の「パッシブな更新 2」[4] は省略した.

x の BFT$[i-1]$ を u に更新する（パッシブ更新）．ノードは，まず挿入時に FFT と BFT のすべてのレベルをレベル 1 からアクティブに更新し，その後は定期的に FFT をアクティブに（BFT はパッシブに）更新する．

ノードの挿入削除がない場合，経路表は各ノードが定期的に更新することでいずれ収束し，FFT（もしくは BFT）のレベル i は，右方向（もしくは左方向）に 2^i 個離れたノードへのポインタを保持する．

ノードがキー k を保持するノードを検索する場合，経路表エントリで右方向で k に最も近いノードへ検索メッセージをルーティングする．受信したノードも同様の処理を繰り返すことで目的ノードに到達する．経路表が収束している場合，最大ホップ数は $\lceil \log n - 1 \rceil$ である．

4.4.2 方針

Kirin の実現方針を以下に述べる．

- Kirin の経路表 (FFT, BFT) エントリには PeerConnection を格納する（FFT[0] と BFT[0] は DDLL_{web} の右リンクと左リンクとする）．基本的に Suzaku と同様の経路表を構築する（挿入時の更新，定期的な更新ともに）が，経路表エントリの各ノードとコネクション（PeerConnection）を確立する必要があるため，経路表の更新シーケンスは Suzaku から変更する．
- Suzaku ではリモートノード障害時に経路表エントリの修復を高速に行うため，経路表エントリにバックアップ用のロケータ集合を保持しているが，WebRTC ではロケータが使えないため，この方式は採用しない．
- Suzaku の各ノード u は，u を削除した後で他のノードが u にメッセージを送信することを抑止するため，u へのポインタを持つノードへのポインタ（リバースポインタ）を保持し，削除時にはリバースポインタを使って u の削除を通知するが，Kirin では削除時にコネクションをクローズすることで相手ノードに削除の通知ができるため，リバースポインタは使用しない．

4.4.3 ノード挿入

ノードを挿入する場合，まず DDLL_{web} によってリングに挿入し（4.3.2 節参照），次に FFT と BFT をレベル 1 から順次アクティブに更新する（FFT と BFT の更新は並行に行う）．それが終わると，後はノードが挿入されている限り FFT をレベル 1 から順次，定期的にかつアクティブに更新する．（紙数の関係でノード削除については省略する．）

4.4.4 Finger Table 更新処理

ノード u が u.FFT$[i]$ $(i > 0)$ をアクティブに更新する処理を述べる（**図 7**．BFT をアクティブに更新する場合は FFT と BFT を読み替える）．なお，ここでは記述を単純化するため，Suzaku の「パッシブな更新 2」に対応する処理と障害時の処理は省略した．

(1) u が `connect` を実行する．Forwarder へのヒントとして，{ 更新レベル (i)，u のキー (ukey)，現在の u.FFT$[i]$ が指すノードのキー (cur)（ただし u.FFT$[i]$ が存在する場合のみ）} を渡す．

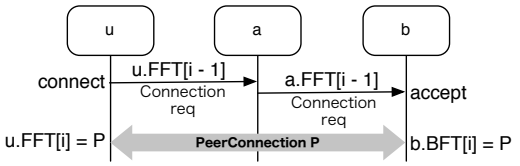

図 7 Finger Table 更新シーケンス例（コネクション確立応答は省略）

(2) Finger Table 更新用 Forwarder は，u.FFT$[i-1]$ が指す PeerConnection を用いてコネクション確立要求を転送する．

(3) コネクション確立要求を受信したノード（a とする）の Forwarder は，a.FFT$[i-1]$ をチェックする．
- (a)（Case 1）空 (null) の場合，`reject` する．
- (b)（Case 2）a.FFT$[i-1]$ の PeerConnection のキーが cur と等しい場合，`reject` する（u が現在の u.FFT$[i]$ が指すノードと再度コネクションを確立することを防ぐため）．
- (c)（Case 3）a.FFT$[i-1]$ の PeerConnection のキーが，ukey と等しいか，ukey を越える場合，リングを一周しているため `reject` する．
- (d) そうではない場合，コネクション確立要求を a.FFT$[i]$ を用いて転送する．

(4) a からコネクション確立要求を受信したノード（b とする）の Forwarder は `accept` を実行し，u とコネクションを確立する．

(5) コネクションが確立したら，u 側では u.FFT$[i]$ を当該 PeerConnection で更新し（アクティブ更新），u.FFT$[i+1]$ の更新に進む．また，b 側では b.BFT$[i]$ を当該 PeerConnection で更新する（パッシブ更新）．

(6) コネクション確立が reject された場合，
- (a)（Case 1 または 3 の場合）FFT 更新終了．
- (b)（Case 2 の場合）u.FFT$[i+1]$ の更新に進む．

4.4.5 Finger Table の障害回復

Finger Table エントリの障害は，WebRTC-Manager からの切断イベントの受信，および送信メッセージに対する Ack メッセージのタイムアウトによって検出する．障害エントリはルーティングには使用しない．このようなエントリはいずれ DDLL_{web} の障害回復処理（レベル 0 の場合）あるいは Finger Table の更新処理（レベル 1 以上の場合）によって取り除かれる．

4.4.6 PeerConnection のクローズ

ノード u の Finger Table エントリが PeerConnection $p1$ から PeerConnection $p2$ に変更された場合を考える．u は以後 $p1$ を必要としないが，$p1$ の相手ノードはまだ $p1$ を使用している可能性があるため，u は単純に $p1$ をクローズす

るわけにはいかない．

このため，自ノードで使用しなくなった PeerConnection に対しては，相手ノードに**クローズ通知**を送って自ノードが当該 PeerConnection を必要としないことを通知し，両側で必要としないことが確認できた時点でクローズする．

5. 実装

WebRTC-Manager，DDLL_{web}，Kirin はすべて TypeScript（JavaScript にコンパイルできる静的型付けが可能な言語）で記述した．外部ライブラリとして，WebRTC を容易に扱うために simple-peer ライブラリ[*3]，WebSocket を扱うために Socket.IO ライブラリ[*4]を用いた．ブラウザノードとポータルノードは同一のコードで動作する．

ソースコードの規模は次のとおりである．WebRTC-Manager（約 1800 行），DDLL_{web}（約 1000 行），Kirin（約 400 行，ただし範囲検索やアプリケーションマルチキャストなどは未実装）．

また，以下の環境で動作を確認した．

- Google Chrome (Windows, Mac, Android)
- Firefox (Windows，Mac)
- Node.js (v6.9.1, v7.5)

6. 評価

ノード数を変化させたときの平均検索時間と平均ホップ数を測定した．ノード数を 1 から 14 まで変化させた．最初のノードはポータルノードとし，2 つ目以降のノードは，すべてブラウザノードの場合とすべてポータルノードの場合で測定した．環境は以下のとおりである．

- MacBook Pro (Early 2013, macOS 10.12.6, Intel Core i5 2.6GHz, 8GB RAM)
- （ブラウザノード）Google Chrome (Mac，バージョン: 60.0.3112.101)
- （ポータルノード）Node.js v6.9.1

本来複数の計算機を用意して測定すべきであるが，環境の準備が困難であったため，すべてのノードは 1 つの計算機上で動作させた．また，すべてのブラウザノードは 1 つの Google Chrome 上で動作させた（1 つのブラウザノードが 1 つのウインドウに対応．この場合でもブラウザノード間の接続には WebRTC が用いられる）．

すべてのノードを挿入し，Finger Table が収束した後，最後に挿入したノードからすべてのノードのキーを検索し，要した時間とホップ数の平均を測定した．検索メッセージとして 1KB のデータを用いた．試行は 10 回行った．

結果を**図 8** に示す．横軸はノード数，破線は平均検索時間（左軸），実線は平均ホップ数（右軸）である．平均ホップ数は対数オーダで増えている（Suzaku と同じ）．平均検索時間は，すべてがポータルノードの場合は対数オーダで増加しているが，ブラウザノードの場合はほぼ線形オーダで増加している．これは，1 つの計算機上で複数のノードを実行しているためであり，ホップ数の推移からも実際の環境では対数オーダで増加するものと考えられる．

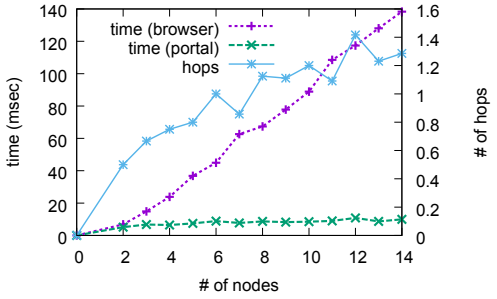

図 8 ノード数と平均ホップ数・平均検索時間の関係

1 つの計算機上で 2 つのブラウザノードを起動しているとき，2 つのブラウザノード間の検索時間（1 ホップ当たりの遅延時間）は約 30.7 ミリ秒であり，実用的な範囲である（なお，実際の環境ではネットワーク遅延時間が加算される）．

7. おわりに

本稿では，WebRTC の特性を考慮したブラウザ間 P2P ネットワークを実現する手法を提案した．また，提案手法を実装したコネクション管理のためのライブラリ WebRTC-Manager およびその上に実装した双方向リング構築アルゴリズム DDLL_{web} とキー順序保存型構造化 P2P ネットワーク Kirin について述べた．

今後の課題としては，Kirin の Finger Table 修復処理の高速化，実ネットワーク環境における Kirin の評価などが挙げられる．

(謝辞) 本研究は JSPS 科研費 JP16K00135 の助成を受けている．

[*3] https://github.com/feross/simple-peer
[*4] https://socket.io/

参考文献

[1] Tsujio, N.: webrtc-chord (online), available from ⟨https://github.com/tsujio/webrtc-chord⟩ (accessed 2015/12/25).
[2] Suchanek, T.: webrtc-kademlia (online), available from ⟨https://github.com/timsuchanek/webrtc-kademlia⟩ (accessed 2016/01/02).
[3] WebTorrent: WebTorrent (online), available from ⟨https://webtorrent.io/⟩ (accessed 2017/06/23).
[4] 安倍広多，寺西裕一：高い Churn 耐性と検索性能を持つキー順序保存型構造化オーバレイネットワーク Suzaku の提案と評価，信学技報，Vol. 116, No. 362 (IA2016-65), pp. 11–16 (2016).
[5] Abe, K. and Yoshida, M.: Constructing Distributed Doubly Linked Lists without Distributed Locking, *Proc. of the IEEE Intl. Conf. on P2P Computing 2015*, pp. 1–10 (2015).
[6] Bu, M. et al.: PEERJS (online), available from ⟨http://peerjs.com⟩ (accessed (2015/12/25)).