ユーザ間データ共有を支援する分散型 Web ブラウザの WebRTC による実装

河野 匠[1]　新城 靖[1]　中村 公洋[1]　三村 賢次郎[1]

概要：分散型 Web ブラウザとは、Web ブラウザ上で動作する協調アプリケーションのための分散 OS である。従来の分散型 Web ブラウザでは、アプリケーション間の通信はブラウザ外の通信路を利用した Remote Procedure Call (RPC) で行っていた。そのため利便性が低く、プログラムの構造が複雑になるという問題があった。本研究では、分散型 Web ブラウザでユーザ間データ共有機能を提供しアプリケーション開発を簡素化する。データ共有として、Access Control List (ACL) を持つファイルシステム、および Operational Transfomation (OT) を用いた Document Object Model (DOM) 木の共有機能を提供する。本分散型 Web ブラウザでは、協調アプリケーションは共有データを更新する部分と読み込み表示する部分で構成され、プログラムの構造が簡素化される。Web ブラウザ間の通信にはブラウザ間で直接通信可能な手法である Web Real-time Communication (WebRTC) を用い、高速化し、利便性を高める。

1. はじめに

現在では多くの情報が Web サイトを通して提供されており、Web ブラウザは必要不可欠なツールとなっている。また、Web ブラウザ上で動作する Web アプリケーションが普及したことにより、今や Web ブラウザはアプリケーションの実行環境、すなわち OS としての側面も持つ。

Web アプリケーションの中には Google Docs や Slack のように、遠隔地に居る人と共同で作業を行うことができるものがある。これを協調アプリケーションと言う。

協調アプリケーションには中央のサーバに依存しているものと利用者間で直接通信を行うものがあり、後者のための基盤として分散型 Web ブラウザが開発された [1]。これは Web ブラウザ上で動作する協調アプリケーションのための分散 OS であり、協調アプリケーションに Remote Procedure Call (RPC) によるブラウザ間の通信機能などを提供する。協調アプリケーションは、この RPC を用いてブラウザで動作するインスタンス間でデータをやり取りする。

協調アプリケーションの中には共同編集を行うアプリケーションのように、高対話型のもの、すなわち、参加しているインスタンス間での細粒度で双方向にデータをやり取りするものがある。しかし、従来の分散型 Web ブラウザではそのようなデータのやり取りを RPC で実装する必要があった。そのようなデータのやり取りには必ずしも RPC は適していない。また、アプリケーションのロジックとデータのやり取りが混在することになり、プログラムの構造が複雑になるという問題がある。

本研究では、このような問題を解決するため、分散型 Web ブラウザでユーザ間データ共有機能を協調アプリケーションに提供する。具体的には、次の 2 種類のデータ共有機能を提供する。

- メール等のメッセージ交換を行うアプリケーションのための Access Control List (ACL) を持つファイルシステム
- 高対話型のアプリケーションのための Operational Transfomation (OT) を用いた Document Object Model (DOM) 木の共有機能

DOM 木とは、HTML 文書をオブジェクトとして扱うためのインターフェースである。また、本分散型 Web ブラウザでは Web ブラウザ間の通信には Web Real-time Communication (WebRTC) を用い、高速化し、利便性を高める。

実装した分散型 Web ブラウザ上で、メッセージを交換するアプリケーションとして電子メールとマイクロブログ、高対話型のものとして日程調整アプリケーションが動作している。これらの協調アプリケーションは共有データを更新する部分と、読み込み表示する部分で構成される。この論文ではユーザ間データ共有機能を使用することによってこのようなアプリケーションのプログラムの構造を簡素化することができることを示す。

[1] 筑波大学

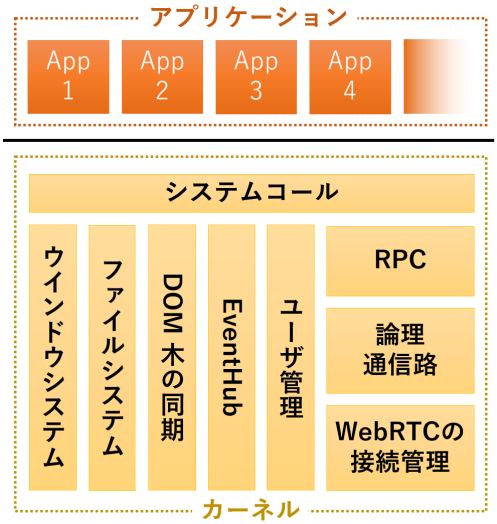

図 1　分散型 Web ブラウザの構成

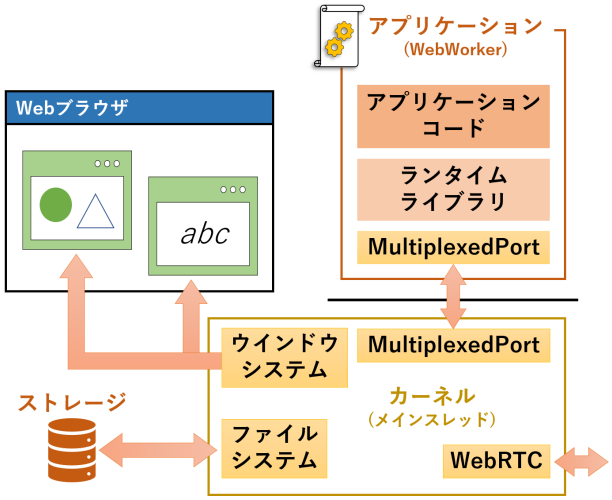

図 2　アプリケーションの構成

2. ユーザ間データ共有を支援する分散型 Web ブラウザ

本研究で提案する分散型 Web ブラウザの構成を図 1 に示す。本分散型 Web ブラウザは、システムコール層、ウインドウシステム、ファイルシステム、DOM 木の同期機能、EventHub、ユーザ管理機能、RPC、論理通信路、および WebRTC の接続管理機能から構成されている。論理通信路は WebRTC の通信路を多重化するものである。RPC は論理通信路を用いて実現する。これらの要素をカーネルと言う。カーネルの上で協調アプリケーションのプロセスが動作する。カーネル、および、アプリケーションは JavaScript で記述される。

2.1 WebWorker によるアプリケーションのプロセスの実装

本分散型 Web ブラウザにおけるアプリケーションの構成を図 2 に示す。各アプリケーションは、ウインドウ・システムのアプリケーションと同様に、Web ブラウザのタブ内にウインドウを持つ。その表示内容は、DOM 木で表現されている。

```
loadApplication(
  "apps/TextEditor.js",
  [ "hello.txt" ]
);
```

図 3　プロセスの生成

本研究ではアプリケーションのプロセスをブラウザの WebWorker[2] 機能を用いて実現する。WebWorker は、Web アプリケーションの処理をバックグラウンドで実行できるようにするための機能である。これを用いることで個々のアプリケーションを別々のスレッドで実行することができる。また、変数も分離されるため、複数の協調アプリケーションの間を隔離し、保護を実現することができる。

現在の Web ブラウザの設計では、WebWorker で実行できることは大きく制限されている。具体的には次ようなことは基本となる JavaScript のコード (メインスレッド) では実行できるが、WebWorker からは実行できない。

- ブラウザが持つ DOM 木へのアクセス
- メインスレッドにある変数へのアクセス

本分散型 Web ブラウザのカーネルはメインスレッドで動いている。メインスレッドと WebWorker スレッドの間でメッセージをやり取りすることができる。そこで本研究では、このメッセージのやり取りを利用して、協調アプリケーションのプロセスからカーネルの機能を利用すること、すなわち、システムコールを提供する。また、シグナルやアップコールと同様に、カーネル側からアプリケーションにイベントの発生などを通知する機構も提供する。

本分散型 Web ブラウザは、プロセスを生成するために、loadApplication() というシステムコールを提供する。図 3 にそのシステムコールの利用例を示す。この例では、“apps/-TextEditor.js” というファイルに含まれる JavaScript コードからプロセスを生成しその引数として “hello.txt” という文字列 (ファイル名) を渡している。

2.2 カーネルのサービスを利用するためのメッセージ送受信

本分散型 Web ブラウザでは、各協調アプリケーションのプロセスは、カーネルが提供するサービスを利用するためにクライアントサーバモデルに基づきメッセージを送受信する。このメッセージの送受信は、MultplexedPort という JavaScript のオブジェクトで実装している。各プロセスは、このオブジェクトを 1 つ持ち、これを使って要求メッセージを送信し、対応する応答メッセージを受信する。

```
let promise = multiplexedPort.request(
  "kernel",
  {
    proc: "createWindow",
    params: {
      title: "Mail Application",
      src: "main.html"
    }
  }
);
...
let result = await promise;
```

図 4 MultiplexedPort による要求メッセージの送信と応答メッセージの受信の例 (ウィンドウの作成)。

```
multiplexedPort.registerHandler(
  "callback",
  (message, reply) => {
    let result = callback_dispatch(message);
    reply(result);
  }
);
multiplexedPort.registerHandler(
  "event",
  (message, reply) => {
    let result = event_dispatch(message);
    reply(result);
  }
);
```

図 5 MultiplexedPort によるコールバックとイベントの受信

図 4 に、MultplexedPort の利用例を示す。この例では、ウインドウを作成するシステムコールである “createWindow” を利用している。request() メソッドは、要求メッセージをサービスを提供しているサーバに送信する。このメソッドの第 1 引数は、何に関する要求なのかを表す service である。この例ではシステムコールの呼び出しであることを表す “kernel” を指定している。第 2 引数は、メッセージの内容である。これには、JavaScript Object Notation (JSON) 形式の連想配列が使われる。呼び出すシステムコールの種類はこの連想配列のキー proc に対応する値として記述する。また、システムコールに与える引数はキー params に対応する値として記述する。キー params の値は連想配列として表現する。

request() メソッドは、返り値として JavaScript の Promise を返す。Promise とは、イベント駆動による非同期的な処理を簡単に記述するための仕組みであり、未来のある時点で値を持つものである [3]。この例では、await を指定して、この結果を取得している。JavaScript では、await を使う方法の他に、then() 関数を利用して受け取る方法もある。

JavaScript におけるプログラミングでは、しばしばコールバックにより結果を受け取ったり、イベントを受け取ることがある。このような場合、プロセスは、要求メッセージを受け取るために MultiplexedPort を逆向きに使う。図 5 にそのような例を示す。registerHandler() は、カーネルから送られてくるメッセージのハンドラを登録するメソッドである。コールバックの呼び出しに用いるメッセージの service は “callback” とする。イベントの送信に用いる要求メッセージの service は “event” である。

2.3 ウインドウ関連のシステムコール

アプリケーションのプロセスは、ブラウザのタブの中にウィンドウを開くことができる。ウインドウの例を図 7 に示す。各ウインドウは最上部にタイトルバーを持ち、その下がウインドウの内容である。ウィンドウは、iframe として実装している。プロセスは、iframe の DOM 木を操作することで、表示内容を変更する。プロセスは、他のプロセスの iframe や iframe の外の DOM 木要素をアクセスしたり操作することはできない。

本分散型 Web ブラウザは、プロセスに対して表 1 に示すようなウィンドウ関連のシステムコールを提供する。createWindow() は、ウィンドウを作成する関数である。作成したウィンドウに対して表示する要素を追加するには、createElement() を用いる。あるいは、setInnerHTML() により、HTML で追加することもできる。プロセスは、listenEvent() を用いて、DOM 木の要素にイベントリスナを設定することができる。

現在のブラウザでは、WebWorker から直接タブに表示されている DOM 木を操作することはできないように設計されている。そのためシステムコールを通して DOM 木の要素を取り扱うときは要素に UUID 形式の識別子を付与し、それを指定する。例えば、createElement を用いて要素を生成すると、戻り値としてその識別子が返される。listenEvent でイベントの監視を行う場合、要素の指定は createElement などから得られた識別子を用いる。

監視しているイベントが発火した場合は、カーネルはアプリケーションに MultiplexedPort を通してメッセージを送信する。このときのメッセージの service は “event” である。そのようなメッセージの内容の例を図 6 に示す。この例は windowId として “mailWindow” を持つウインドウのある要素で click イベントが発生したことを表している。

本研究の分散型 Web ブラウザは、ウインドウマネージャを持つ。これはタイトルバーやウインドウの枠を生成する。ユーザは、それらを用いてウインドウの移動や大きさの変更を行う。

アプリケーションはウインドウマネージャを通してタイ

```
{
  windowId: "mailWindow",
  id: "94b65c86-fae6-11e9-8f0b-362b9e155667",
  type: "click"
}
```

図 6　ウィンドウシステムから送信される、イベントを表すメッセージの例

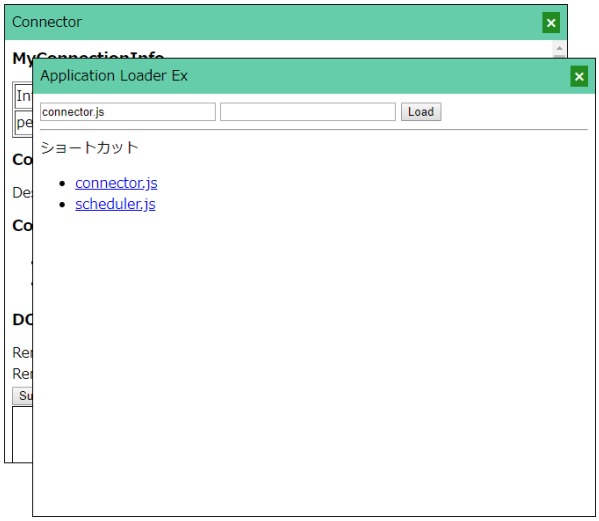

図 7　ウインドウの例 (アプリケーションローダ)

トルバーにボタンを設置することができる。アプリケーションはこのボタンを通してユーザによる操作を受けることができる。例えばダイアログのウインドウに閉じるボタンを設置し、ユーザによるウインドウを閉じる操作を受け付けることができる。図 7 ではタイトルバーの右側に閉じるボタンを設置している。

2.4　EventHub

協調アプリケーションはファイルの更新があったことやリモートユーザがオンラインになったことなどを、その時点で即座に知りたいことがある。そのため、本分散型 Web ブラウザはイベントを発行したり、発行されたイベントを購読するための機構として EventHub を提供する。

EventHub はシステムコールとして subscribeEvent を提供する。このシステムコールを使用すると、アプリケーションは EventHub に発行されたイベントを購読することができる。イベントが発行された場合、その通知はアプリケーションに MultiplexedPort のメッセージとして送信される。メッセージの service は “dbrowser-event” である。各イベントは module、type、および payload から構成されている。module には “WindowSystem” や “UserManager” など、イベントを発行する分散型 Web ブラウザの構成要素名を指定する。type には “Disconnected” や “Modified” など、何が発生したのかを指定する。payload にはその通知

```
{
  module: "File",
  type: "Modified",
  payload: "/sampleApp/illust1.bmp"
}
```

図 8　EventHub 経由で送信される、イベントを表すメッセージの例

の詳しい内容を記述する。購読者は購読時にモジュール名とタイプを指定することでイベントをフィルタすることができる。

アプリケーションに送信されるメッセージの例を図 8 に示す。この例は “/sampleApp/illust1.bmp” というパス名を持つファイルが更新されたことを表している。

3.　ユーザ管理機能と WebRTC による高速なブラウザ間 RPC の実装

3.1　ユーザ管理機能

分散型 Web ブラウザはユーザ管理機能を提供する。ユーザ管理機能は分散 OS としてユーザの情報を持ち、分散型 Web ブラウザの起動時にユーザの認証を行う。

各ユーザは自身以外の複数のユーザと協調作業を行う。これらの自分以外のユーザのことを本研究では友人と言う。ユーザ管理機能は各友人について、以下の情報を保持している。

- UUID による識別子
- ユーザ名
- 認証に必要な情報 (公開鍵など)
- そのユーザが実行している分散型 Web ブラウザのインスタンスへの通信路
- 以下で述べる共通ランデブ・ポイント・インターフェースにおけるユーザの識別子

ユーザ管理機能は新たに友人との通信路が開設された場合の相手の認証も行う。各協調アプリケーションは Access Control List (ACL) によるアクセス制御に、通信相手の友人の識別子やユーザ名を利用できる。

本分散型 Web ブラウザでは、ユーザ管理機能を共通ランデブ・ポイント・インターフェース (Common Rendevou Point Interface, CRPI)[4] を用いて実装する。CRPI は、インスタント・メッセンジャやインターネット上のファイル同期アプリケーションを使って認証されたメッセージ送受信機能を提供するものである。個々のメッセージ送受信の仕組みをランデブ・ポイントと呼ぶ。CRPI は、各ユーザの PC で動作し、HTTP、および、WebSocket で機能を提供する。

CRPI は、次のような静的な情報を保持する。

- コンタクトリスト。ユーザの識別子、スクリーン名、グループ。

表 1　ウインドウ関連のシステムコール

名称	用途	主要な引数	引数の説明	戻り値
createWindow	ウインドウの生成	title	ウインドウタイトル	なし
createElement	DOM 木の要素の生成	tagName	要素名	要素の識別子
setInnerHTML	要素の子要素を HTML 文書で指定	elementId	listenEvent を参照	なし
		value	HTML 文書	
listenEvent	DOM 木の要素のイベントの監視	elementId	イベントを監視する要素の識別子	なし
		eventName	イベント名	

- ユーザごとのランデブ・ポイント。

CRPI は、次のような動的な情報を送受信する機能を提供する。

- 特定のユーザに対して短いメッセージを送信する。
- コンタクトリストにある全ユーザに、短いメッセージをマルチキャストする。
- 認証されたユーザから短いメッセージを受信する。

現在、本分散型 Web ブラウザでは、次の 2 種類のランデブ・ポイントを利用している。

- インスタントメッセンジャ Tox を使うもの。通信相手は、Tox の ID で指定される。Tox の ID は、公開鍵より作られる。
- ファイル同期アプリケーション Syncthing[5]、または、Resilio Sync[6] を使うもの。通信相手ごとに事前に 2 つの鍵を交換し、2 つのディレクトリを共有する。各ノードは 1 つのディレクトリをメッセージの送信用、もう 1 つを受信用に利用する。

従来の分散型 Web ブラウザでは、このようなブラウザ外の、認証された安全な通信路をそのまま利用して協調アプリケーションを実装していた。たとえば、インスタント・メッセンジャ Skype や XMPP サーバを使っていた [1]。しかしながら、このようなブラウザ外の通信の仕組みは性能が低く、Web ブラウザで動作するビデオ通話アプリケーションのように、高い性能を必要とするものでは利用できない。また、Web ブラウザから直接利用するには、プラグインを必要とするため利便性が低い。

本研究では、このようなブラウザ外の通信をそのまま利用するのではなく、以下の節で述べる WebRTC の通信路を確立するためのブートストラップの仕組みとして用いる。ブラウザの内部では WebRTC の利用により、高速で利便性が高い通信路を実装する。

3.2　共通ランデブ・ポイント・インターフェースによる WebRTC のシグナリング

本分散型 Web ブラウザではブラウザ間の通信路は、WebRTC を用いて Web ブラウザ同士を Peer-to-peer (P2P) で接続することによって形成する。これが様々な通信の基礎となる通信路となる。一度確立された WebRTC の通信を利用すれば、中央サーバの障害の影響を受けず、サーバ管理者により通信内容が監視されることはない。

WebRTC の通信路を確立するために、2 つのブラウザは IP アドレスやポート番号などの接続情報が含まれている Interactive Connectivity Establishment (ICE) Candidates と呼ばれるメッセージや、受け入れ可能なメディアの情報が含まれている Session Description Protocol (SDP) 形式のメッセージをやり取りする必要がある。このメッセージのやり取りをシグナリングと言う。このようなシグナリングの手段は WebRTC の規格では定められていない。

本分散型 Web ブラウザにおける 2 ノード間の WebRTC のシグナリングは、3.1 節で述べた共通ランデブ・ポイント・インターフェース (CRPI) を用いて、以下の手順で行われる。

(1) 2 ノードを Offerer と Answerer に分ける。
(2) Offerer で SDP 形式の Offer メッセージを作成し、LocalDescription として保存する。
(3) Offerer は Offer メッセージを Answerer に送信する。
(4) Answerer は Offer メッセージを受信し、RemoteDescription として保存する。
(5) Answerer はそれをもとに SDP 形式の Answer メッセージを作成し、LocalDescription として保存する。
(6) Answerer は Answer メッセージを Offerer に送信する。
(7) Offerer は Answer メッセージを受信し、RemoteDescription として保存する。
(8) Offerer、Answerer 共に ICE Candidates を生成し、お互いに送受信する。
(9) 受信した ICE Candidates を保存する。

CRPI を通してメッセージを送受信する様子を図 9 に示す。図 9 では Alice の使用している分散型 Web ブラウザから Bob の使用している分散型 Web ブラウザに向けて SDP または ICE のメッセージが送信されている。このメッセージの送信には 3.1 節で述べた「特定のユーザに対して短いメッセージを送信する」機能を使用する。

メッセージを送信する Alice 側ではまず、メッセージの宛先について、分散型 Web ブラウザにおけるユーザの識別子から CRPI におけるユーザの識別子に変換する。そして、CRPI に対して、変換後のユーザの識別子を使ってメッセージを送信するように要求する。この要求は CRPI が HTTP で提供している WebAPI を通して行われる。こ

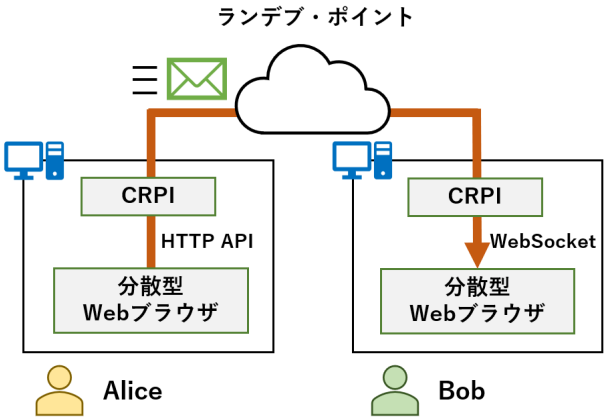

図 9　CRPI を通してメッセージを送受信する様子

の要求を受けた CRPI は予め指定されたランデブ・ポイントに向けてメッセージを送信する。

一方 Bob 側では、CRPI に WebSocket で接続することでメッセージを受信する。Bob 側の CRPI が Alice からのメッセージを受信すると、そのメッセージを Bob の使用している分散型 Web ブラウザに WebSocket を通して送信する。このとき、メッセージの送信元は CRPI によって認証されており、メッセージが Alice からのものであることが保証されている。最後にメッセージの送信元の ID を CRPI におけるユーザの識別子から分散型 Web ブラウザにおけるユーザの識別子に変換することで、Bob はメッセージの受信を完了する。

3.3　論理通信路

基礎となる WebRTC の通信路を形成した後、本研究では、その通信路の中に複数の論理通信路を形成できるようにする。これにより、アプリケーションが複数の種類の通信を同時に行うことが容易になり、また、通信路の開設要求ごとに WebRTC のシグナリングという重たい処理を行う必要がなくなる。また、WebRTC の通信路が一時的に切断された場合、再接続を試みたり、メッセージの再送信を行うこともできる。

論理通信路は JavaScript のクラス LogicalConnection として実装する。このクラスは、基礎となる WebRTC の通信路を示す RTCDataChannel と同じメソッドを持つ。したがって、従来の WebRTC のアプリケーションは、このクラスを利用してそのまま動作する。

図 10 に論理通信路を形成してデータを送信する例を示す。LogicalConnection のコンストラクタは第 1 引数に WebRTC のデータチャネル、第 2 引数に論理通信路の識別子をとる。図中の underlyingConnection は接続済みの WebRTC のデータチャネルを表している。コンストラクタで論理通信路の識別子を指定しなかった場合、論理通信路の識別子は自動的に設定される。

図 11 に論理通信路からデータを受信する例を示す。データを受信する場合はデータを受信したい論理通信路の識別子を指定する。

```
let logicalConnection = new LogicalConnection(
  underlyingConnection,
  "connection-12345"
);
logicalConnection.send(
  "Hello World!"
);
```

図 10　論理通信路を形成してデータを送信する例

```
let logicalConnection = new LogicalConnection(
  underlyingConnection,
  "connection-12345"
);
logicalConnection.setReceiver(
  (data) => {
    doSomething(data);
  }
);
```

図 11　論理通信路からデータを受信する例

3.4　WebRTC 上の RPC の実装

本研究の分散型 Web ブラウザは、協調アプリケーションに対してノード間通信の仕組みとして RPC を提供する。RPC は下位層の通信路として 3.3 節で述べた論理通信路を用いる。論理通信路の識別子は “RPC” とする。

図 12 にサーバにおいて RPC の要求メッセージのハンドラを登録する例を示す。この例は下位層の通信路の開設時に呼ばれる関数を表している。RPC のハンドラの登録には registerApp() メソッドを用いる。registerApp() メソッドには第 1 引数としてアプリケーション名、第 2 引数としてアプリケーションのバージョン番号、第 3 引数として要求メッセージのハンドラを与える。このハンドラには引数として要求メッセージと、応答メッセージを返すための関数が与えられる。図 12 の例ではアプリケーションとしてファイルサーバを登録している。要求メッセージは連想配列として表現され、キー op の値としてファイルシステムに対する操作が記述される。それ以外のキーの値は各操作に与える引数として扱われる。例えば操作 “readFile” は引数としてキー path の値をとる。

図 13 に RPC のクライアントの例を示す。要求メッセージの送信には sendRequest() メソッドを用いる。sendRequest() メソッドには第 1 引数としてアプリケーション名、第 2 引数としてアプリケーションのバージョン番号、第 3 引数として要求メッセージを与える。sendRequest() メソッドは戻り値として Promise を返す。この Promise に

```
function onConnectionEstablished(
  friendId,
  underlyingConnection
)
{
  let logicalConnection =
    new LogicalConnection(
      underlyingConnection,
      "RPC"
    );
  let rpc = new RPC(logicalConnection);
  let fs = getLocalFS();

  rpc.registerApp(
    "remotefs", 1,
    (msg, reply) => {
      switch(msg.op) {
        case "readFile":
          let result = fs.readFile(msg.path);
          reply(result);
        case "writeFile":
          ...
      }
    }
  );
}
```

図 12 RPC のサーバにおける要求メッセージのハンドラの例 (ファイルサーバ)

```
let bobLogicalConnection =
  getLogicalConnection("Bob");
let bobRPC = new RPC(bobLogicalConnection);

let result = await bobRPC.sendRequest(
  "remotefs", 1,
  { op: "readFile", path: "/document/test.txt" }
);
```

図 13 RPC のクライアントの例 (ファイルの読み出し)

await を利用することによって、RPC の結果を得ることができる。

4. ユーザ間データ共有機能

4.1 ファイルシステム

本分散型 Web ブラウザはユーザ間データ共有機能として、ファイルシステムを提供する。このファイルシステムはローカルからだけでなく、RPC を通してリモートからアクセスすることができる。また、ACL を持ち、ユーザ管理機能で認証されたユーザの情報を用いてアクセス制御を行う。

本ファイルシステムは下位層に BrowserFS [7] を用いて実現する。BrowserFS はブラウザ上でファイルシステムを実現するライブラリであり、ブラウザのストレージ機能である LocalStorage や IndexedDB をバックエンドとして用いる。本研究では IndexedDB をバックエンドとして用いる。

```
{
  owner: "a558b0f2-018f-11ea-8d71-362b9e155667",
  createDateTime: "2019-10-30T05:23:28.182Z",
  accessDateTime: "2019-10-31T06:33:49.725Z",
  modifyDateTime: "2019-10-31T06:34:01.842Z",
  acl: [
    {
      id: "a558b0f2-018f-11ea-8d71-362b9e155667",
      r: true, w: true, x: true
    },
    {
      id: "cfc88ac4-018f-11ea-a69e-362b9e155667",
      r: true, w: true, x: true
    }
  ]
}
```

図 14 ファイルのメタデータの例

4.1.1 ファイルのメタデータ

BrowserFS で扱うことができるメタデータは非常に限られている。そのため、本研究ではファイルのメタデータを別途保持する。本研究のメタデータは作成時刻、アクセス時刻、更新時刻、所有者、および ACL から構成される。

ファイルメタデータの例を図 14 に示す。ACL のエントリはそのエントリの対象となるユーザの識別子、R、W、および X から構成されている。R、W、および X は true もしくは false の値をとる。本ファイルシステムの ACL における R、W、および X の意味は Unix のパーミッションと同一である。true がビットが立っている状態、false が立っていない状態に対応する。ただし、ディレクトリにおいては X を使用するが、ファイルにおいては X を使用しない。どのエントリの対象にもならないユーザは、(R,W,X)=(false,false,false) となる ACL エントリを持っているものとして取り扱う。つまり、そのファイルやディレクトリに対する一切の操作を行うことはできない。

ファイルのメタデータは AppleDouble[8] と同様に、対象となるファイルの名前の末尾に.meta を付与し、下位層 (BrowserFS) のファイルに保存する。例えば、/sampleDir/test.txt というパスを持つファイルのメタデータを/sampleDir/test.txt.meta というファイルに保存する。ディレクトリの場合はディレクトリ内に.meta という名前のファイルを作成し、これに保存する。例えば、/sampleDir というパスを持つディレクトリのメタデータをファイル/sampleDir/.meta に保存する。

```
let fs = getLocalFileSystem();
fs.watch(
  "/tweetscache/alice",
  (path, op) => {
    switch(op) {
      case "added":
        ...
      case "removed":
        ...
      case "modified:
        ...
  }
);
```

図 15 ファイルを監視するコードの例

4.1.2 ファイルシステム関数

本ファイルシステムが提供する主要な関数を表 2 に示す。これらの機能はアプリケーションに対してシステムコールとして提供される。ファイルが存在しなかったり、アクセス権がない操作等でエラーが生じたときには例外を発生させる。

4.1.3 ファイル・ディレクトリの監視

本ファイルシステムはローカルのファイルやディレクトリに対する操作を監視し、通知する機能を提供する。この機能を利用すれば、例えばローカルの別のアプリケーションがファイルを変更したとき、それをイベントの発生として知ることができる。ディレクトリに対して監視を行った場合、ディレクトリへのファイルの追加や削除の操作、および既にディレクトリ内に存在するファイルに対して変更の操作が行われたことを知ることができる。

図 15 にファイルを監視するコードの例を示す。fs はローカルのファイルシステムを表すオブジェクトである。この例ではディレクトリ “/tweetscache/alice” を監視している。watch は引数として操作が行われたときに呼び出される関数をとる。この関数は呼び出されたときに引数として操作が行われたファイルのパスである path と操作の種類である op が与えられる。op の取りうる値は added、removed、および modified の 3 つで、それぞれ追加操作、削除操作、変更操作に対応している。

4.2 ファイルの単方向同期

本分散型 Web ブラウザはユーザ間データ共有機能の一つとしてファイルの単方向同期機能を提供する。この機能はリモートファイルの内容をローカルに複製するものである。ただし、メタデータのうち ACL は複製しない。ファイルの単方向同期はディレクトリ単位で行う。リモートディレクトリに更新 (ファイルの追加・削除、およびファイルの変更) があった場合、ローカルディレクトリの内容を更新する。アプリケーションは 4.1.3 項で述べたディレクトリの監視機能を利用して、更新が発生したことを知ることができる。

```
syncDirectory(
  "Alice",
  "/tweets",
  "/tweetscache/alice",
  1800
);
```

図 16 ファイルの単方向同期の開始

現在の実装ではリモートディレクトリの更新は、ポーリングによって監視している。ポーリングの間隔としては利用可能な資源量によるが、5 分から 30 分程度を想定している。

ファイルの単方向同期のためのシステムコールを表 3 に示す。図 16 にファイルの単方向同期を開始するコードを示す。この例では、単方向同期を開始するシステムコールである syncDirectory を使用して、30 分ごとにユーザ “Alice” の “/tweets/” というディレクトリの内容を、ローカルの “/tweetscache/alice” というディレクトリに同期している。syncDirectory は戻り値として同期の識別子を返す。同期を停止する場合はこの識別子を引数としてシステムコール stopSync を呼び出す。

4.3 DOM 木の同期

4.1 節で述べたファイルシステムは、電子メールやマイクロブログのように、ユーザが作成したコンテンツを保存するようなアプリケーションには適している。4.2 節で述べたファイル同期機能はポーリング間隔として 5 分から 30 分程度を想定しており、協調編集のような高対話型の協調アプリケーションには適していない。

そこで本分散型 Web ブラウザでは、高対話型の協調アプリケーションのために、高速なデータ構造の同期機能を提供する。データ構造としては、DOM 木を用いることにした。その理由は、JavaScript で広く一般的に利用できるからである。

DOM 木の同期には Operational Transformation (OT) [9] を用いる。これはデータの一貫性を保つためにある操作を別の操作へ変換する手法である。本研究では DOM 木用の OT ライブラリである dom-ot[10] を用いる。dom-ot では、メッセージのトランスポートを置き換え可能になっている。本研究では、それを 3.3 節で述べた WebRTC の論理通信路を用いて実装する。

表 4 に DOM 木の同期のために分散型 Web ブラウザが提供するシステムコールを示す。DOM 木の同期には MasterDocument と EditableDocument の 2 種類のオブジェク

表 2　ファイルシステムが提供する主要な関数

名称	用途	主要な引数	引数の説明	戻り値
readFile	ファイルの読み込み	path	ファイルのパス	ファイルの内容
writeFile	ファイルの書き込み	data	書き込むデータ	なし
mkdir	ディレクトリの作成	path	作成するディレクトリのパス	なし
readdir	ディレクトリに含まれている ファイルとディレクトリの 列挙	path	列挙するディレクトリのパス	ディレクトリに含まれている ファイルとディレクトリの 名前の配列
setACL	ファイルやディレクトリへの ACL の設定	path	ACL を設定するファイルや ディレクトリのパス	なし
		ACL	設定する ACL	
setDateTime	ファイルやディレクトリが 持つ各種時刻の設定	create	作成日時	なし
		access	アクセス日時	
		modify	更新日時	
getACL	ファイルやディレクトリ に設定された ACL の取得	path	取得する ACL が設定された ファイルやディレクトリの パス	ACL
getDateTime	ファイルやディレクトリが 持つ各種時刻の取得	path	時刻を取得するファイルや ディレクトリのパス	作成日時、アクセス日時 および更新日時が含まれた 連想配列

表 3　ファイルの単方向同期のために提供するシステムコール

名称	用途	引数	引数の説明	戻り値
syncDirectory	単方向同期を開始する	remoteId	同期元ディレクトリを 保有している友人の識別子	同期の識別子
		remotePath	同期元ディレクトリのパス	
		localPath	同期先ディレクトリのパス	
		interval	更新を検出する間隔 (秒) 最低でも 10 以上の値が必要	
stopSync	単方向同期を停止する	id	停止する同期の識別子	なし

トを使用する。MasterDocument は同期においてコーディネータとなるオブジェクトである。EditableDocument は DOM 木の変更を差分として送信したり、送信されてきた差分を DOM 木に適用したりするオブジェクトである。DOM 木の同期を行うときは 1 つの MasterDocument と複数の EditableDocument をスター状に接続する。

図 17 に同期のコーディネータである Alice が実行するコードを、図 18 に同期へ参加する Bob が実行するコードを示す。このコードは同期したい DOM 木の親要素と EditableDocument を結びつけ、MasterDocument と EditableDocument を接続する。図 17 では MasterDocument がローカルにあるため、connectDocument の引数の to.name は null になっている。このコードは同期の準備と同様に bindDocument を行い、リモートにある MasterDocument に対して connectDocument を行っている。同期からの離脱を行うときは disconnectDocument を用いて接続を切断する。

```
let mDocId = createMasterDocument(domTreeParent);
let eDocId = bindDocument(domTreeParent);
connectDocument({
  from: eDocId,
  to: {name: null, id: mDocId}
});
```

図 17　DOM 木の同期の準備 (コーディネータ)

```
let ownerId = getUserId("alice");
let eDocId = bindDocument(domTreePlaceholder);
connectDocument({
  from: eDocId,
  to: {name: ownerId, id: mDocId}
});
```

図 18　DOM 木の同期への参加

5. アプリケーション

本分散型 Web ブラウザの機能を評価するために、本研究では次のようなアプリケーションを実装した。

- メール
- マイクロブログ
- 日程調整

表 4　DOM 木の同期のために提供するシステムコール

名称	用途	引数	引数の説明	戻り値
createMasterDocument	MasterDocument を作成する。	element	初期値として設定する要素	作成された MasterDocument の ID
		documentId	[任意] 使用したい MasterDocument の ID	
bindDocument	EditableDocument を作成し与えられた要素と結びつける。	element	作成される EditableDocument と結びつけられる要素	作成された EditableDocument の ID
connectDocument	EditableDocument を MasterDocument に接続し同期を開始する。	from	接続元の EditableDocument の ID	成功した場合は true それ以外は false。
		to.name	[任意] 接続先のユーザ ID	
		to.id	接続先の MasterDocument の ID	
disconnectDocument	接続を切断し同期を終了する。	documentId	同期を終了する EditableDocument の ID	成功した場合は true それ以外は false。

5.1　メール

本研究では、ファイルシステムを利用して、電子メールのアプリケーションを実装する。本研究のメールは認証された宛先ユーザに向けて、サーバを通さずブラウザノード間で直接転送される。メールの受信者も発信者を認証できる。

図 19 にメールアプリケーションの構造を示す。各ユーザは、自分のブラウザのファイルシステム中にメールの受信箱となるディレクトリを用意する。他人の受信箱のディレクトリへのファイルの保存をメールの送信、自身の受信箱となるディレクトリにあるファイルの読み出しをメールの受信とする。例えば、Alice が Bob に向けてメールを送信する場合、Alice は Bob のファイルシステムに遠隔からアクセスし、受信箱となるディレクトリにメールを保存する。他人にメールを読まれるのを防ぐため、受信箱となるディレクトリは、受信箱のオーナーだけが読み書き可能で、他の人は、ファイルの書き込みのみ可能なように ACL を設定する。具体的には、受信箱のオーナーに対しては (R,W,X)=(true,true,true) とし、それ以外の人に対しては (R,W,X)=(false,true,true) とする。また、複数のユーザが同じ名前のファイルを作成することを避けるため、保存するファイル名にはランダムな文字列を付与する。

図 19 は Alice と Bob がメールを送りあっている様子を表している。実線はメールデータの受け渡しが伴う操作を表している。破線はメールデータの受け渡しが伴わない操作を表している。

メールアプリケーションはメール送信機能、メール受信機能、およびユーザインターフェースから構成される。メール送信機能はユーザインターフェースからのメール送信操作に応じて、ユーザインターフェースに入力されたメールの宛先や内容を読み取り、メールを作成する。そして、作成したメールを宛先の受信箱に保存する。

メール受信機能はユーザインターフェースからのメール受信操作に応じて、ローカルの受信箱からメールを読み取る。そして、読み取ったメールをユーザインターフェースに表示する。

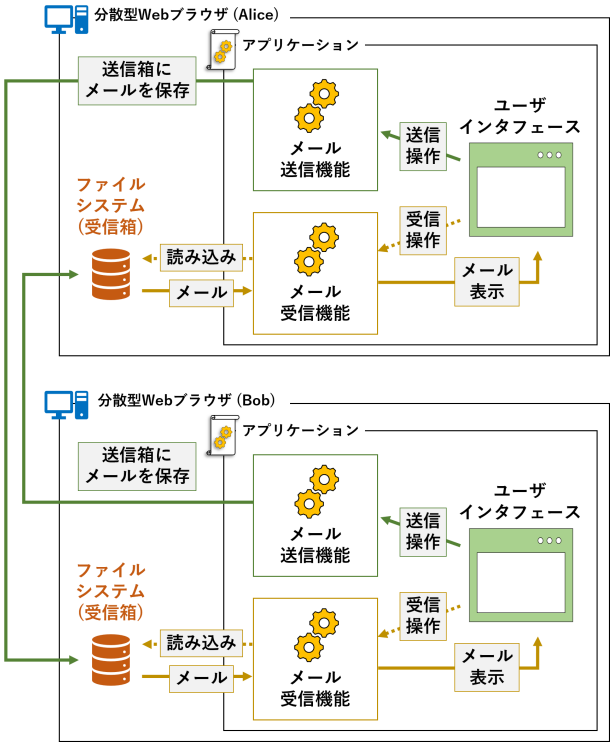

図 19　メールアプリケーションの構造

本メールアプリケーションには、受信したメールの内容に応じて外部のアプリケーションを実行し、パラメータを渡す機能がある。詳しくは 5.3 節で述べる。

5.2　マイクロブログ

本研究で提供するファイルの単方向同期機能を用いて Twitter のようなマイクロブログアプリケーションを実装した (図 20)。マイクロブログアプリケーションでは、ユーザは日常生活での出来事など、様々なことを短いメッセージや写真で投稿する。また、ユーザは他人のメッセージに

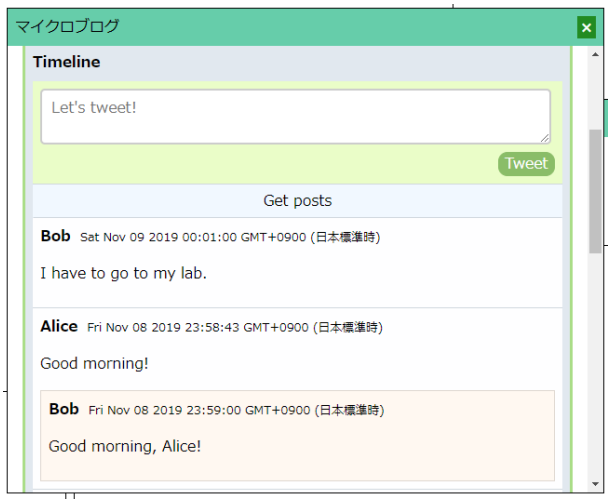

図 20　マイクロブログアプリケーション

図 22　日程調整アプリケーション

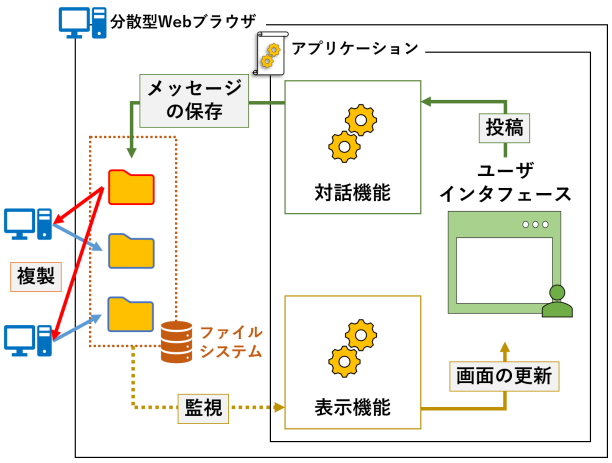

図 21　マイクロブログアプリケーションの構造

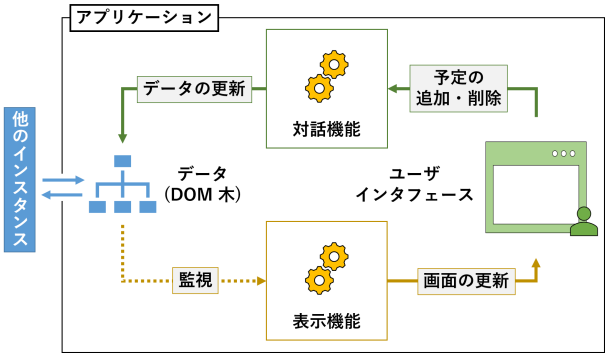

図 23　日程調整アプリケーションの構造

対してコメントを追加することができる。

図 21 にマイクロブログアプリケーションの構造を示す。アプリケーションは対話機能と表示機能から構成される。前者はローカルのユーザと対話し、投稿操作に応じてローカルのディレクトリに自身のメッセージを保存する。また、4.2 節で述べたファイルの単方向同期機能を用いて、友人がメッセージを保存したディレクトリをローカルのディレクトリに複製する。図の左端にある、分散型 Web ブラウザの外部からディレクトリを指している矢印はこの複製を表している。また、同じく左端にある、ディレクトリから分散型 Web ブラウザの外部を指している矢印は、友人の分散型 Web ブラウザによる自身のディレクトリの複製を表している。後者はローカルのディレクトリおよび、分散型 Web ブラウザによって友人と同期されているディレクトリを監視し、新たなメッセージが追加されたら画面に表示する。

5.3　日程調整アプリケーション

DOM 木の同期機能を用いた協調アプリケーションの例として、本研究では日程調整アプリケーションを実装した (図 22)。このアプリケーションでは、各利用者はある行事の複数の開催候補日について、参加できるかどうかを記入する。アプリケーションは行事の開催候補日に対応したカレンダーと各々の参加者の出席できない日を表示する。利用者が参加可否を記入した場合、そのインスタンスは自分の持っている DOM 木に参加可否の情報を入れる。すると、DOM 木の同期が発生し、他の利用者においても参加可否が表示される。

日程調整アプリケーションではユーザ間のデータの同期に基づいた設計を行ったため、図 23 のようにアプリケーションの構造が単純になった。このコードは、対話機能と表示機能から構成される。前者はローカルのユーザと対話し、それに応じて DOM 木を書き換える。後者は DOM 木を監視し、それが変更されたら画面に表示する。このとき、変更はローカルのユーザによるものか、リモートのユーザによるものかを区別することはない。

5.1 節で述べたメールアプリケーションを日程調整に招待する手段として使用することができる。メールアプリケーションは受け取ったメールに含まれた特殊な文字列を解釈する。具体的にはメール中の invite: から始まる文字列を解釈し、アプリケーションを起動するリンクへと変

表 5　実験環境

	コンテナ 1	コンテナ 2
CPU	Intel Core i7-9700K	
メモリ	32GB	
OS	Ubuntu 18.04	
ブラウザ	Google Chrome Version 77.0.3865.90 (Official Build) (64-bit)	
インターネット アクセスライン	SINET5 (大学ネットワーク経由)	インターリンク (B フレッツ経由)
ping	9 [ms]	
iperf	100〜450 [Mbps] (方向・時間帯によって大きく変動する)	

換する。このリンクをクリックすると対応するアプリケーションが起動され、起動時の引数として DOM 木の ID 等が与えられる。日程調整アプリケーションはこの引数を用いて DOM 木の同期へ参加することができる。

6. 性能

本研究で実装した分散型 Web ブラウザが実用可能な性能を持つかどうかを確認するために実験を行った。実験環境の詳細を表 5 に示す。実験環境を Docker を用いて構築した。同一ホスト上にコンテナを 2 つ作成し、両コンテナの上で本研究の分散型 Web ブラウザを動作させた。両コンテナ間の通信はインターネットを経由するように設定した。性能を測定する実験では、プログラムを 6 から 100 回繰り返し実行し、メッセージ転送時間や実行時間の最小値、中央値、および最大値を測定した。

6.1 CRPI を用いた Offer メッセージの受け渡しにかかる時間

CRPI を用いて WebRTC のシグナリングを行い、単一のメッセージの受け渡しにかかった時間を測定した。ここでは、受け渡しにかかった時間を送信側が CRPI にメッセージを送信し始めてから、受信側が CRPI からメッセージを受信し終わるまでの時間とする。

実験結果を表 6 に示す。インスタントメッセージングサービスである Tox をランデブ・ポイントとして用いた場合は中央値で 0.21 秒しかかかっていない。これは分散型 Web ブラウザのユーザにストレスを与えることのない、シグナリングを行うには十分な性能である。一方で、ファイル同期アプリケーションである Syncthing を用いた場合は中央値で 10.15 秒と、体感で遅いと感じるほどの時間がかかっている。しかし本分散型 Web ブラウザは、通信路の開設は起動時に一度だけ行い、分散型 Web ブラウザを終了させるまで維持する。そのためアプリケーションの実行前には通信路は開設された状態であり、シグナリングによる待ちは生じない。よって、本分散型 Web ブラウザにおいて Syncthing をランデブ・ポイントとして用いる妨げにはならない。

表 6　WebRTC のシグナリングにおける単一のメッセージの受け渡しにかかる時間 (CRPI 経由)

	ランデブ・ポイント	
	Tox	Syncthing
最小値 [s]	0.13	7.73
中央値 [s]	0.21	10.15
最大値 [s]	0.29	10.88

表 7　RPC を用いたエコーにかかる時間

	データのサイズ	
	200 [B]	100 [kB]
最小値 [ms]	12	275
中央値 [ms]	14	343
最大値 [ms]	23	1391

6.2 RPC の基本性能

分散型 Web ブラウザの RPC 機能の性能を簡単なエコーバックを行うサービスを用いて測定した。2 つのコンテナ上の分散型 Web ブラウザのうち、片方でサーバ、片方でクライアントを実行した。エコーバックに与えるデータとして 200 B と 100 kB のランダムなデータを用意した。200 B のデータは、マイクロブログの典型的なメッセージサイズ、100 kB のデータはマイクロブログで使われる写真のサイズを想定したものである。

表 7 にこの呼び出しにかかった時間を示す。200 B の小さなメッセージでは、表 5 の ping の測定値よりも中央値で 5 ミリ秒大きくなっているが、実用上十分な性能があると言える。また、写真を想定した 100 kB のメッセージでも中央値で 343 ミリ秒で往復できている。

6.3 遠隔にあるファイルシステムに対する readFile() にかかる時間

分散型 Web ブラウザのファイルシステムの性能を測定した。6.2 節の実験と同様に 2 つのコンテナ上の分散型 Web ブラウザ間のうち、片方でサーバ、片方でクライアントを実行した。そして、クライアントからサーバにあるファイルを readFile() を用いて読み込んだ。読み込むファイルとして 200 B と 100 kB の 2 種類の、ランダムなデータが含まれたファイルを用意した。

表 8 に読み込みにかかった時間を示す。このように実験で用いた環境では、マイクロブログのテキストメッセージを想定した 200 B のファイルを 85 ミリ秒で、写真を想定した 100 kB のファイルを 187 ミリ秒で読み込むことができた。

ファイルを大量に使用するアプリケーションとして 5.2 節のマイクロブログアプリケーションが挙げられる。マイクロブログではメッセージのサイズが小さく、数が多い。ここであるユーザの 1 時間あたりのメッセージ数を 20、

表 8　遠隔にあるファイルシステムに対する readFile() にかかる時間

	ファイルサイズ	
	200 [B]	100 [kB]
最小値 [ms]	36	135
中央値 [ms]	85	187
最大値 [ms]	95	222

表 9　DOM 木の同期の反映にかかる時間

	要素のサイズ	
	200 [B]	100 [kB]
最小値 [ms]	14	117
中央値 [ms]	14	252
最大値 [ms]	19	1225

メッセージ 1 つのサイズを 200 B とすると、表 8 より、メッセージの同期は 1 時間あたり 5 秒程度で終わると期待される。これは 1 時間に対して十分小さい値である。分散型 SNS では、1 人のユーザが 100 人程度の友人と交流できれば十分である。複数の友人のメッセージを同期する場合でも、1 時間当たり 1 分程度で終了すると期待される。このことから、本分散型 Web ブラウザのファイルシステムの性能は分散型 SNS のマイクロブログというアプリケーションについては十分実用的であると言える。

6.4　DOM 木の同期の反映にかかる時間

分散型 Web ブラウザの DOM 木の同期機能の性能を測定した。6.2 節の実験と同様に 2 つのコンテナ上の分散型 Web ブラウザ間で、5.3 節で述べた日程調整アプリケーションと類似のプログラムを走らせた。片方ノードで DOM 木へ要素を追加してからもう片方でこの変更が反映されるまでの時間を測定した。DOM 木の要素として、ランダムなデータを 200 B 分持つ要素と、100 kB 分持つ要素の 2 種類の要素を用意した。

表 9 に同期の反映にかかった時間を示す。このように実験で用いた環境では、200 B の要素を 14 ミリ秒で、100 kB の要素を 252 ミリ秒で同期させることができた。この結果から、5.3 節で述べた日程調整のようなアプリケーションでは、DOM 木の同期が実用的な時間で行えることが分かる。

7.　関連研究

Browsix[11] は Web ブラウザ上で C、C++、および、Go 言語で記述された単一 PC 用アプリケーションを動作させる試みである。アプリケーションの例として dash(シェル)、GNU Make、pdfLaTeX、bibtex を使用可能にしている。Browsix はアプリケーションに対して、プロセス、ファイルシステム、パイプ、およびソケットといった Unix 系 OS が提供してきた機能を提供する。そのため、既存の単一 PC 用アプリケーションを改変することなく動作させることができる。プロセスは WebWorker を用いて実現し、システムコールの呼び出しはメインスレッドと WebWorker 間のメッセージのやり取りによって実現している。本研究では、Browsix を大いに参考にして分散型 Web ブラウザを実装した。Browsix と比較して、本研究の特徴は複数のノードで動作する協調アプリケーションの開発と実行を支援している点である。例えばユーザ間ファイル同期機能を使用すると、アプリケーション開発者は簡単に協調アプリケーションを作成することができる。

Voutilainen ら [12] は、実行中の JavaScript のアプリケーション (プロセス) を状態の同期により、あるコンピュータから別のコンピュータへ移送する研究を行っている。この研究におけるアプリケーションの状態とは、Web アプリケーションにおける JavaScript の変数と DOM 木のことを指す。著者らは DOM 木をそのまま同期することが現実的ではないとして、Virtual DOM に基づいた同期方法を提案している。一方本研究で実装した DOM 木の同期機能ではアプリケーション全体の移送ではなく協調作業に焦点をあてている。そのため、他のノードと限られたデータのみを同期する。また、一度だけの移送ではなく、データを継続的に同期する点も異なる。

これまでに Web ブラウザを用いた協調作業が研究されてきた。Opera Unite は Web ブラウザでサーバを動作させることができる [13]。例えば、ユーザはファイルサーバを走らせ、友人とファイルを共有することができる。Opera Unite ではブラウザ間通信は Opera 社の中央サーバを必要とする。

論文 [14] と [15] では、ユーザが PC で Web サーバを動作させ、友人にカメラなどのデバイスを使用させられるようにすることを提案している。しかし、これらの研究は Web サーバとブラウザ間のセキュアで認証された通信チャネルの確立をサポートしていない。

OP [16] と Gazelle [17] はマイクロカーネルアーキテクチャに基づいて設計された Web ブラウザである。このアーキテクチャでは Web ブラウザは、信頼できる小さなカーネルを通してメッセージの交換を行う、複数の信頼できないモジュールの集まりである。このマイクロカーネルはブラウザ間通信をサポートしていない。

P2P ファイル共有を用いると中央サーバを必要とせずにファイルを配布することができる。IPFS[18]、Dat[19]、Syncthing[5] のように、Web ブラウザを通したファイルアクセスが可能なものもある。これらのシステムは不特定多数の人に対して Web コンテンツを提供することができるが、ユーザの認証や、それに基づいたアクセス制御を行うことができない。本分散型 Web ブラウザは、アクセス制御を行うことができるファイルシステムを提供する。

8.　まとめ

本論文ではユーザ間データ共有を支援する分散型 Web

ブラウザについて述べた。分散型 Web ブラウザとは、Web ブラウザ上で動作する協調アプリケーションのための分散 OS である。各アプリケーションは DOM 木を表示できるウインドウを持つ WebWorker のプロセスである。ブラウザ間の通信は WebRTC を用いることで中央サーバに依存することなく実現する。ユーザ間データ共有としてリモートアクセスが可能なファイルシステム、および、ファイルの単方向同期機能を提供する。また、高対話型協調アプリケーションのために DOM 木の共有機能を提供する。

本研究のユーザ間データ共有機能を用いることで、協調アプリケーションを簡素に実装することができる。本研究では、具体的なアプリケーションとしてメール、マイクロブログ、および、日程調整のアプリケーションを実装した。これらの実装を通して、ユーザ間データ共有機能の有用性を確認した。また、高速なインターネット回線を用いた実験により、その性能が実用的であることを確認した。

現在の分散型 Web ブラウザは 1 人のユーザが 1 台のデバイスを利用することを想定している。今後、複数のデバイスで分散型 Web ブラウザを動作させられるようにし、ファイルやユーザ情報などの共有も含め、デバイス間の連携を行えるようにしたい。また、効率の良い単方向同期の実装と多数のノードを利用した性能測定も今後の課題である。

参考文献

[1] Shinjo, Y., Guo, F., Kaneko, N., Matsuyama, T., Taniuchi, T. and Sato, A.: A distributed web browser as a platform for running collaborative applications, *7th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2011)*, pp. 278–286 (2012).

[2] HTML Living Standard, Web workers, https://html.spec.whatwg.org/multipage/workers.html. Accessed: 2019-11-06.

[3] ECMAScript 2015 Language Specification, https://www.ecma-international.org/ecma-262/6.0/#sec-promise-objects. Accessed: 2019-11-06.

[4] 蛸井博，新城靖，佐藤聡，中井央：分散型 SNS のための共通ランデブ・ポイント・インターフェースの実装，情報処理学会コンピュータシステム・シンポジウム論文集，Vol. 2016, pp. 110–120 (2016).

[5] Syncthing, https://docs.syncthing.net/. Accessed: 2019-11-08.

[6] Resilio Sync, https://www.resilio.com/individuals/. Accessed: 2019-11-08.

[7] Vilk, J. and Berger, E. D.: Doppio: Breaking the Browser Language Barrier, *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 508–518 (2014).

[8] Faltstrom, P., Crocker, D. and Fair, E. E.: RFC 1740 - MIME Encapsulation of Macintosh files - MacMIME, https://tools.ietf.org/html/rfc1740. Accessed: 2019-11-08.

[9] Ellis, C. A. and Gibbs, S. J.: Concurrency Control in Groupware Systems, *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data (SIGMOD '89)*, pp. 399–407 (1989).

[10] Klehr, M.: dom-ot, https://github.com/marcelklehr/dom-ot. Accessed: 2018-10-09.

[11] Powers, B., Vilk, J. and Berger, E. D.: Browsix: Bridging the Gap Between Unix and the Browser, *the 22th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*, pp. 253–266 (2017).

[12] Voutilainen, J., Mikkonen, T. and Systä, K.: Synchronizing Application State Using Virtual DOM Trees, *Current Trends in Web Engineering (ICWE 2016)*, pp. 142–154 (2016).

[13] Leichtenstern, T.: File sharing with Opera 10.10 Unite, *Linux Magazine (Feburuary 2010)*, pp. 40–42 (2010).

[14] Lin, K., Chu, D., Mickens, J., Zhuang, L., Zhao, F. and Qiu, J.: Gibraltar: Exposing Hardware Devices to Web Pages Using AJAX, *Proceedings of the 3rd USENIX Conference on Web Application Development (WebApps12)*, pp. 75–87 (2012).

[15] Lyle, J., Nilsson, C., Isberg, A. and Faily, S.: Extending the Web to Support Personal Network Services, *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pp. 711–716 (2013).

[16] Grier, C., Tang, S. and King, S. T.: Secure Web Browsing with the OP Web Browser, *2008 IEEE Symposium on Security and Privacy*, pp. 402–416 (2008).

[17] Wang, H. J., Grier, C., Moshchuk, A., King, S. T., Choudhury, P. and Venter, H.: The Multi-principal OS Construction of the Gazelle Web Browser, *Proceedings of the 18th Conference on USENIX Security Symposium*, pp. 417–432 (2009).

[18] Benet, J.: IPFS - Content Addressed, Versioned, P2P File System, http://arxiv.org/abs/1407.3561, *The Computing Research Repository (CoRR)* (2014).

[19] Robinson, D. C., Hand, J. A., Madsen, M. B. and McKelvey, K. R.: The Dat Project, an open and decentralized research data tool, *Scientific Data*, Vol. 5, No. 180221 (2018).