修士論文

REST と WebSocket アプリケーションの統合モデルの提案と検証

Proposal and Verification of a Model Integrating REST and WebSocket Applications

青山学院大学大学院 理工学研究科 理工学専攻 知能情報コース

Intelligence and Information Course

Graduate School of Science and Engineering

Aoyama Gakuin University

石畑 翔平

Shohei, Ishihata

目 次

図 目 次

表 目 次

第1章　はじめに

1.1　研究背景

近年 Web 上の情報量，更新頻度が高まっている．そして，SNS の登場によりその方向性は加速する一方である．そこで，データの更新をリアルタイムに反映するための通信技術として WebSocket が開発された．WebSocket は以前の擬似的なリアルタイム通信と比べ，レイテンシ，オーバヘッド，実装の簡易性に優れている．しかし，WebSocket にはいくつかの解決すべき課題が存在する．その一つが WebSocket アプリケーションのリソース指向アーキテクチャの構築である．

今まで多くの Web アプリケーションは REST に沿って開発されてきた．REST はリソース指向を実現するためのアーキテクチャスタイルである．REST に沿ったアプリケーションは RESTful Web アプリケーションと呼ばれ，拡張性，汎用性，簡易性に優れる．しかし，ステートフル通信である WebSocket は REST に対応していない．これは Web サービスとしての性質を失い，多くの利点を損なうことを意味する．そこで本研究ではこの問題を解決するために REST と WebSocket アプリケーションの統合モデルを提案する．

1.2　研究目的

本研究の目的は，WebSocket アプリケーションのリソース指向への対応である．そのため，本研究では REST と WebSocket アプリケーションとの統合モデルを提案する．統合モデルはリソース指向に WebSocket アプリケーションを実装するための指針となる．本統合モデルを適用することで Web アプリケーションとして性質を取り戻し，URI によるリソースの参照，状態の復帰などが可能となる．これらはアプリケーションの動作を単純化させ，ユーザビリティを向上させる．本研究では，実際に様々な WebSocket アプリケーションで統合モデルの検証を行い，実装方法，利点などを示している．

1.3 本論文の構成

本論文の構成は以下の通りである．本論文は本章を含め全 6 章で構成される．本章は，研究背景と目的について述べた．第 2 章は，本研究で扱う WebSocket に関連する技術と REST やその他のアーキテクチャに関連する基礎技術を解説している．第 3 章は，WebSocket のプロトコルやその拡張，関連するフレームワークやライブラリについて解説する．また，現在対応が必要である WebSocket の課題について述べる．第 4 章は，WebSocket の課題を解決する REST と WebSocket アプリケーションの統合モデルの提案を行う．第 5 章は，オセロや株価チャートアプリケーションなど，いくつかの具体的なアプリケーションにより本統合モデルの検証を行う．最後に，第 6 章は本研究の成果と今後の課題についてまとめている．

第2章　基本技術

2.1　TCP

2.1.1　概要

TCP (Transmission Control Protocol) [2] とは，TCP/IP [3] のトランスポート層にあたり，UDP [4] (User Datagram Protocol) と共にネットワークの通信において重要な役割を担っている通信技術である．TCP はコネクション型の信頼性のある Byte Stream Service を提供するのが特徴である．主に文章の転送などパケット損失が重要な問題となる通信に使われる．WebSocket は単一の TCP コネクションを利用して通信を行う．

2.1.2　TCP/IP

TCP/IP とは，インターネットを実現するための様々なプロトコルの総称である．表 2.1 のように上位層からアプリケーション層，トランスポート層，ネットワーク層，リンク層の 4 階層に分かれている．上位層は下位層を利用することで実現する．WebSocket はアプリケーション層に属する．階層化によって各プロトコルの独立性を向上させ，設計を容易にしている．例えば，データの転送はトランスポート層が保証しているため，アプリケーション層はデータの転送漏れなどを考慮する必要がない．

表 2.1: TCP/IP の階層とプロトコル

階層名	プロトコル例
アプリケーション層	`HTTP`, `WebSocket`
トランスポート層	`TCP`, `UDP`
ネットワーク層	`IP`, `ICMP`
リンク層	`Ethernet`, `PPP`

2.1.3 TCP の通信

TCP は UDP に比べ，レイテンシは大きいが，信頼性が高いという特徴を持つ．また，接続はすべてのデータが転送されるまで確立され，サーバのリソースを占有し続ける．図 2.1 は TCP コネクションの開始から終了のイメージである．

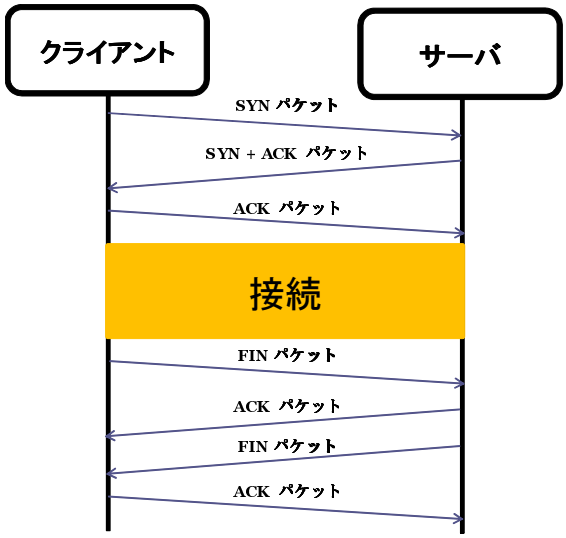


図 2.1: TCP コネクション

接続

TCP コネクションはスリーウェイハンドシェイクという方式で接続が行われる．接続手順は次のようになる．

1. クライアントが接続先のサーバに SYN (synchronize) パケットを送信する．SYN パケットはコネクション確立要求のためのパケットで，サーバのポート番号や，クライアントの初期シーケンス番号が入っている．

2. SYN パケットを受け取ったサーバは，サーバの初期シーケンス番号を含む SYN パケットで応答する．また，SYN パケットに加え，クライアント側の初期シーケンス番号+1 の確認応答番号を含んだ ACK (acknowledge) パケットを送信する．ACK パケットはコネクションを認めたことを意味する．

3. クライアントはサーバから送られてきたSYNパケットに対して，サーバの初期シーケンス番号+1の確認応答番号を含んだACKパケットを送信する．

以上の三つのフェーズによって接続を確立し，この確立した通信路をVirtual Circuitと呼ぶ．

通信

コネクションの接続後はTCPによるデータの通信が行われる．TCPは一つのパッケトを送るごとに，ACKパケットを返す．そのため，一定時間ACKパケットが送られてこないとデータの送信者は再び同じパケットを送信する．これによってデータの損失を防ぐのである．その他にもフロー制御や，順不同で到着したパケットの再整列などを行い，信頼性のある通信を行う．

切断

すべてのパケットが送受信されると接続を終了する．図2.1はクライアントからTCPコネクションを切断しているが，切断はサーバ，クライアント双方から開始することができる．TCPコネクションの切断は以下の通りの手順で行われる．

1. サーバにコネクションの終了を要求するFINパケットを送信する．FINパケットはfinishの最初の三文字を取ってFINパケットという．接続の終了を表す．
2. FINを受け取ったら，それに対するACKパケットを送信する．
3. サーバのアプリケーションが終了したら，それを知らせるFINパケットを送信する．
4. サーバ側からのFINパケットに対して，ACKパケットを送信する．

以上の四つのフェーズでコネクションを切断する．

2.1.4 ポート番号

TCPはアプリケーションによってポート番号を使い分けている．ポート番号は1から65535番まであり，1から1023番がサーバアプリケーション用に予約されている．WebSocketはHTTPのポートがそのまま使用されるため80番である．ポート番号をアプリケーションごとに分けることで，多重接続を可能にしている．

2.2 URI

2.2.1 概要

URI (Uniform Resource Identifier) [5] [6] とは，リソースを一意に識別するためのルールである．識別には ASCII 文字が使われるため，アルファベットや数字，一部の記号しか使用することができない．そのため，ASCII 以外の文字を使用する場合はパーセントエンコーディングが必要である．また，IRI (Internationalized Resource Identifier) [7] を使うことで，多言語化することも可能である．

現在 Web では，リソースを識別するために URI が使われている．リソースとは Web 上のありとあらゆる情報を指している．例えば人といった物理的なものから，友達や友情といった概念まで様々である．そして、URI を使用してリソースの通信を行う．通信に使用する技術は，URI スキームで指定され，通常 Web では `http` スキームが使用される．異なるスキームも利用することが可能で，例えば WebSocket の場合 `ws` が使用される．他にも `mailto` や `ftp` など様々なスキームがある．公式の URI スキームの一覧は IANA (Internet Assigned Number Authority) で確認することができる [8].

2.2.2 URI の構文

URI の構文は以下のように構成されている．

```
URI = scheme "://" host ":" port "/" path [ "?" query ] [ "#" fragment ]
```

まず URI はスキームから開始される．ここでリソースにアクセスするための手法を識別する．例えば HTTP 通信ならスキームは `"http"` となる．次に，リソースの場所を表すためのホスト名やポート番号，パスが続く．ホスト名はリソースを持っているコンピュータの名前を指している．IP アドレスによる指定も可能だが，FQDN (Fully Qualified Domain Name) [9] [10] を用いる方が望ましい．FQDN とは，ドメイン名，サブドメイン名，ホスト名を省略せずに全て指定した記述形式のことである．ホスト名とポート番号は `":"` で区切る．ポート番号は通常省略され，予約されたポート番号が使用される．

その後，`"/"` で階層化されたパスが続く．パスはリソースの場所を示すため，構造化された予想可能なパスが好ましい．例えば 2013 年のログを `"log/2013/"` というパスで公開している場合，2014 年のログのパスは `"log/2014/"` とするべきである．続いて，`"[]"` は任意であることを意味する．任意の値の一つが `"query"` である．クエリはパスと `"?"`

で区切られ，指定したアプリケーションにデータを渡す場合などに使用される．クエリは複数指定可能で，その場合は "&" で連結される．クエリの後に続くのが "fragment" である．フラグメントはページ内リンクなど，リソース内部の細かい部分の指定に使用される．以上が URI の構文である．

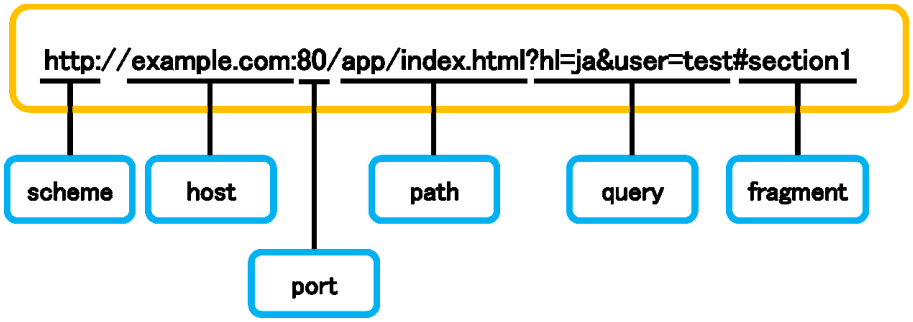


図 2.2: URI の例

図 2.2 は実際の URI の使用例である．"http" はスキームである．HTTP で通信していることがわかる．"example.com"，"80" はホスト名とポート番号である．省略した場合でも，HTTP 通信なので自動的に 80 番が使用される．続いてリソースの場所を示すためのパスが指定される．"index.html" は省略可能である．そして，クエリ，フラグメントと続く．このように，Web 上にあるあらゆるリソースを URI で指定することが可能である．

2.2.3 URI によるアドレス可能性

URI は Web の発展に大きく貢献している．URI は Web の簡易性や汎用性を向上させ，あらゆる Web サービスに必須の技術となっている．この URI が可能にする重要な特徴の一つがアドレス可能性である．アドレス可能性とは，操作するリソースの情報が URI に含まれている状態である．アドレス可能性は多くのことを実現する．

例えば，Google で WebSocket を検索すると "https://www.google.com/search?q=WebSocket" という URI が与えられる．そのため，私たちは論文や本，Web ページなどにこの URI を参照させることができる．しかし，URI が与えられていない場合，論文の読者に WebSocket を検索させるためには，「google.com に接続し，検索バーで WebSocket と入力し，検索ボタンを押す.」と指示しなければならない．これはとても不便である．

アドレス可能でない例としては Google Maps (https://maps.google.co.jp/) がある．Google Maps は代表的な Ajax アプリケーションである．そのため，ユーザはページを遷

移することなくマップ上を移動することができる．これはとても魅力的であるが，アドレス可能性を損なってしまう．それはリソースの状態が変化しても，URI が更新されないことが理由である．地図を動かしても URI が変わらないため，ユーザは URI で現在の地図を共有することができない．アドレス可能性には URI によるリソースの識別が必須なのである．

アドレス可能性の利点は他にもある．例えば電子メールによる Web ページの共有である．アドレス可能でない場合，HTML ファイルをダウンロードして，添付しなければならない．また，ブックマークを可能にし，URI によるキャッシュを可能にする．これらはすべて URI によってリソースを一意に識別しているためである．

2.3 HTTP

2.3.1 概要

HTTP (Hypertext Transfer Protocol) [11] [12] は元々ハイパーテキストの転送用プロトコルとして策定された技術である．しかし，現在は XML などのハイパーテキストに限らず，画像や動画，音声，JavaScript プログラムなど，様々なリソースの転送に利用され，現在の Web に必須の技術となっている．TCP/IP のアプリケーション層に位置する．通信は各リクエストが独立した，ステートレスでシンプルな通信が行われる．しかし，様々な拡張が存在し，ステートフルな通信も可能である．例えば Cookie [13] や，サーバと接続を維持し続ける Comet [14] などがある．ただ，HTTP はステートレスを前提にしているため，ステートフル通信はパフォーマンスの低下に繋がりやすく，注意が必要である．また，ステータスコードも様々なものが規定されていて，細かく指定されている．

2.3.2 歴史

HTTP は IETF (Internet Engineering Task Force) で 1996 年に初めて標準化され，RFC 1945 [15] として公開された．このときのバージョンが 1.0 である．しかし，それ以前でも HTTP は使われていたため，便宜的に HTTP1.0 以前のバージョンを HTTP0.9 と呼んでいる．HTTP0.9 は Web を発明した Tim Berners Lee が文章を転送する目的で策定したプロトコルである．現在のような HTTP ヘッダは存在せず，メソッドも GET のみの単純な通信技術であった．その後，NCSA Mosaic や Netscape Navigator，Internet Explorer などの Web ブラウザが登場したことで，HTTP が使われ始め，様々な機能が追加されて

いった．しかし，HTTP1.0 は各 HTTP ベンダの様々な利害関係から独自の機能が実装され，相互運用性の低いものとなってしまった．そのため，RFC 1945 はインターネット標準ではなく，カテゴリが Informational として位置づけられている．

そこで，これらの問題を解決するため策定されたのが HTTP1.1 である．1997 年に RFC 2068 [16]，1999 年に RFC 2616 [11] が公開された．HTTP1.1 では HTTP1.0 に，チャンク転送やコンテントネゴシエーション，キャッシュコントロール，持続的接続など様々な機能が追加されている．HTTP1.1 は現在も使用されていて，単に HTTP という場合は HTTP1.1 を指す．また，その後も HTTP に関する議論は続けられていて，HTTP Bis [17] と呼ばれる様々なドラフトが公開されている．

2.3.3 リアルタイム Web への対応

様々な拡張や性能の向上のための議論が行われている HTTP だが，リアルタイム Web へ対応するための技術も存在する．それが Polling や Long Polling である．それぞれの仕組みについては第 3 章で解説する．これらの技術は，レイテンシが大きいという HTTP の課題を解決するために開発された．特に Long Polling は通常の HTTP 通信に比べ，レイテンシの面を大きく改善している．しかし，これらの技術はオーバヘッドを大きくし，スケーラビリティの面で大きな課題がある．これらを解決するために作られたのが WebSocket である．

2.3.4 HTTP メソッド

現在 HTTP で定義されるメソッドは全部で 8 つである．定義されているメソッドとその役割が表 2.2 である．その中でも一般的に使われるメソッドは `GET`，`POST`，`PUT`，`DELETE` である．表 2.2 のメソッドはすべて使用可能だが，使用するメソッドを少なくすることでクライアントとサーバの通信をシンプルにし，汎用性の高いアプリケーションを作成することができる．また，この 4 つのメソッドを正しく使用することが Web アプリケーションには求められる

GET

`GET` は指定した URI のリソースを取得するのに使用する．HTTP で最も利用頻度の高いメソッドで，唯一 HTTP0.9 から存在するメソッドである．また，`If-Modified-Since`

表 2.2: HTTP メソッド

メソッド名	説明
GET	指定した URI のリソースを取得
POST	指定した URI にリソースを作成
PUT	指定した URI のリソースを更新
DELETE	指定した URI のリソースを削除
OPTIONS	サポートしているメソッドの問い合わせ
TRACE	接続経路の調査
CONNECT	プロキシにトンネリングを要求
HEAD	リソースのヘッダを取得

や，If-Unmodified-Since，If-Match，If-None-Match，If-Range ヘッダーフィールドをリクエストに付加することで，GET リクエストに条件をつけることも可能である．例えば If-Modified-Since は，指定した時刻以降に更新があるかないかを調べるヘッダである．この時，更新がない場合は，サーバはボディ無しでレスポンスを返す．このように条件を指定することで，キャッシュの有効活用やネットワーク帯域を節約が可能になる．

POST

POST はリソースの作成に使用される．GET に次いで利用頻度の高いメソッドである．リクエストには作成したいディレクトリを URI で指定し，ボディに内容を入れる．そして，リソースの作成が成功した場合，レスポンスにリソースの作成を意味するステータスコード 201 と作成されたリソースの URI を Location ヘッダに入れて返す．つまり，リソースの URI を指定するのはサーバ側で，クライアントは親ディレクトリの指定のみである．

通常 POST はリソースの作成に使用されるメソッドであるが，様々な動作に使われる可能性がある．例えばリソースへのデータの追加や，他のメソッドでは実現できない機能などを担当する．そのため，他のメソッドとは違い，リクエストを見ただけではどういう動作を行うかを判別することができず，サーバの実装に依存する．それゆえ，POST はとても多機能で便利であるが，使い方によってはアプリケーションの可視性，汎用性が低くなる可能性があるため，使用には注意が必要である．

PUT

PUT はリソースの更新に使われる．URI を指定し，リクエストボディに変更内容を入れる．レスポンスにはボディを含めても，含めずにボディがないことを意味するステータスコード 204 を返しても良い．この際，指定した URI にリソースが存在しない場合，リソースの更新ではなく作成となる．リソースの作成の場合は，レスポンスにリソースの作成を意味するステータスコード 201 を返す．

上記のように POST で通常行うリソースの新規作成を PUT で行うことも可能である．POST と PUT の違いは，PUT はリソースを作成する際に URI を指定する点である．POST の場合はサーバが URI を決定する．そのため，PUT ではサーバの状態をクライアントが把握している必要がある．例えば，URI にユーザ ID を含む場合，PUT はサーバに同じユーザ ID が存在しないかを把握してから，リクエストを出す必要がある．そのため，特別な理由がない場合，リソースの作成は POST で行い，URI はサーバが決定する方が好ましい．

DELETE

DELETE はリソースの削除に使われる．DELETE のレスポンスはボディを持っていないため，ステータスコードは動作が成功したことを 200 だけでなく，ボディがないことを意味する 204 が使われることもある．

2.4 REST

2.4.1 概要

REST (Representational State Transfer) とは，Roy Fielding が 2000 年に発表した論文 [18] の中で提案された分散ハイパーメディアシステムのためのソフトウェアアーキテクチャスタイルである．Web のアーキテクチャスタイルは他にも SOAP [19] などいくつか存在するが，REST は現在の Web アーキテクチャスタイルの主流となっている．例えば Google API [20] や，Twitter API [21] などは REST の考えが導入されている．このように，REST に沿ったアプリケーションを RESTful アプリケーションと呼ぶ．高いユーザビリティ，高い汎用性から多くのアプリケーションが RESTful に実装されている．

しかし，RESTful は REST を完全に満たしているわけではない．例えば，REST ではステートレス性は必須の制約だが，RESTful は実用性から cookies によるセッション管理を認めている．RESTful は REST の考えを反映したものではあるが，実用性をより重視

したものとなっている．そのため，何を RESTful とするかは様々な議論があり，やや曖昧である．そこで本論文では，RESTful の基準の一つとして ROA を取り上げる．

2.4.2 REST

Roy Fielding が論文の中で提案した REST は，HTTP など使用する技術を限定しておらず，いくつかのアーキテクチャから構成されている．提案されたアーキテクチャは，クライアント・サーバシステム，ステートレス，キャッシュ，統一インタフェース，階層化システム，コードオンデマンドの 6 つである．これらのアーキテクチャをすべて満たしているものが REST である．

クライアント・サーバシステム

端末をクライアントとサーバの二つの役割に分割する分散型のコンピュータシステムである．サーバはアプリケーションのリソースを管理し，サービスを提供する役割である．それに対してクライアントは，画面の表示などのユーザインタフェースを担当し，サービスを利用する役割を担う．ユーザインタフェースをクライアントが担当することで，サーバに PC やスマートフォン，ゲーム機器など多様な端末でアクセスすることを可能にしている．また，アプリケーションのほとんどの処理をサーバが担うため，サービス管理が容易でクライアント側の負担が小さいなどの利点がある．例えば，Google などの検索エンジンを使用することで，クライアントは自身の端末にデータをインストールせずに，大量の Web ページにアクセスすることが可能である．これはサーバがリソースをすべて管理し，クライアントはそれにアクセスするだけで利用可能なためである．しかし，これにはサーバに集中的に負荷がかかってしまうという欠点も存在する．DoS 攻撃 (Denial of Service attack) [22] は，この問題を利用してサーバのリソースを枯渇させる攻撃である．そのため，アプリケーション開発には十分なサーバのリソースの確保が必要となる．

ステートレス

ステートレスとは，サーバがクライアントの状態を管理しない通信方法である．クライアントの状態を管理しないことで，サーバは各通信ごとにリソースを解放し，他のクライアントにリソースを使用することができる．これによって，サーバの負荷を軽減し，スケーラビリティを向上させることができる．また，各通信が簡略化されるため，実装も容

易である．しかし，ステートレスでサーバがクライアントの状態を知るためには，クライアントに通信のたびに情報を送信させなければならない．そのため，ログインなどの機能を実装する場合，オーバヘッドが大きく，セキュリティ面に問題を生じる．

キャッシュ

キャッシュとは，一度取得したリソースをクライアントが再利用する仕組みである．キャッシュを利用することで，サーバとクライアント間の通信を減らし，ネットワーク帯域を節約することができる．また，データの再取得の必要がないため処理を高速化することができる．しかし，キャッシュが古くなることで正しい情報を得られなくなる可能性がある．

統一インタフェース

統一インタフェースとは，リソースの操作を統一された限定的なインタフェースで行うアーキテクチャのことである．Roy Fielding はこの統一インタフェースを REST の中心的な特徴と位置づけている．統一されたインタフェースを使う利点は，サーバとクライアントの独立性を高め，設計をシンプルにすることができる点である．これによって，サーバやクライアントが変わっても，常に同じインタフェースが使われることが保証される．現在の多様なサーバやクライアントの実装は，統一インタフェースが可能にしている．

階層化システム

階層化システムとは，システムをいくつかの階層に分割するアーキテクチャのことである．階層化することで設計がシンプルになり，スケーラビリティが向上する．例えばクライアントとサーバの間にアクセス制限を行うためのプロキシを追加するなど，必要に応じて追加・削除を行う．このような階層化システムが実現できるのは，インタフェースが統一されているためである．しかし，階層化システムはオーバヘッドやレイテンシが大きくなるといった欠点も存在する．

コードオンデマンド

コードオンデマンドとは，クライアントがサーバからプログラムコードをダウンロードし，実行するアーキテクチャである．Roy Fielding は唯一コードオンデマンドを REST でオプション扱いにしている．例えばコードオンデマンドの技術としては JavaScript や

Flash などが挙げられる．コードオンデマンドを使う利点は後からクライアントの機能を拡張できる点である．しかし，クライアントによってはその技術が使えない可能性や，アプリケーションの可視性が低下するなどの問題がある．

2.4.3 ROA

ROA (Resource Oriented Architecture) とは，RESTful Web Service [23] の中で Leonard Richardson，Sam Ruby によって提案された RESTful を定義するための基準である．REST はアーキテクチャスタイルなため抽象性が高く，実際にサービスを実装する段階になると幅広い解釈が可能である．そのため，RESTful の基準はやや曖昧となってしまっている．ROA はこの曖昧な RESTful に一つの明確な基準を与えるのが目的である．ROA が明言する基準とは統一インタフェース，アドレス可能性，ステートレス性，接続性である．ROA はあくまで一つの基準であるため，RESTful アプリケーションは ROA を無視することも可能である．しかし，RESTful の目的であるスケーラビリティやユーザビリティの向上において ROA はとても有効であり，アプリケーションを実装する際に考慮すべきルールである．

統一インタフェース

REST ではインタフェースを特に明示していない．重要なことはインタフェースが統一していることである．これは ROA でも同様であるが，通常 Web では HTTP が使用される．そして，その中でも GET，POST，PUT，DETETE の 4 つのメソッドが主に使用される．ROA では使用するメソッドはこれらに限定するべきで，新しいメソッドを作成することに反対している．新しいメソッドは特定のサービスでしか使用できないため，インタフェースの統一性を損なってしまう．

さらに統一インタフェースは，使用するメソッドを限定するだけでなく，正しく使用することが重要である．例えば GET はリソースを取得するためのメソッドである．それに対して，図 2.3 では GET にパラメータを渡すことで削除を行おうとしている．このような使用方法は不正であり，RESTful とは言えない．不正な使用はスケーラビリティを損なうばかりか，セキュリティの面でも問題ある．そのため，ROA ではメソッドの限定と正しい使用を定義している．

図 2.3: GET の不正な使用

アドレス可能性

RESTful にとってアドレス可能性はとても重要なルールである．2.2.3 で述べたように，アドレス可能性はユーザビリティを高め，サービスの簡易性を高める．アドレス可能性とは，URI のみで操作するリソースを識別できる特性である．そのため，URI は同時に二つ以上のリソースを表すことはできない．ただし，リソースが二つ以上の URI を持つことは可能である．この際，URI はシンプルで構造的な，変更されにくい URI [24] が推奨されている．

ステートレス性

ステートレス性は簡易性やスケーラビリティを向上させる．そのため，スケーラビリティやユーザビリティを重視している REST において重要なルールとなっている．しかし，現在は cookies を使用したセッション管理が主流になっており，ROA も cookies の使用は認めている．そのため，cookies による通信は完全なステートレスではないが，RESTful である．ただし，ROA はクライアントが cookies の値を管理できる使用方法のみに限定している．つまり，サーバが cookies の値を指定し，クライアントの状態を管理する cookies の使用法は RESTful でない．その理由としては，この方法ではサーバが何をしているのかをクライアントが知ることができず，クライアントにとって大きな不利益となるためである．cookies を使用する場合はクライアントが値を定め，クライアントの状態を送るための手段として使用されなければならない．

接続性

接続性とは，URI によってリソースが相互接続されている性質である．接続性はユーザのサービスの操作を容易にし，リソースの関係を把握するのに有効である．また，接続性はユーザビリティを向上するだけでなく，開発者にとっても有益である．図 2.4 の左は接

続性のないサービス，右は接続性のあるサービスである．図 2.4 のようにハイパーリンクを使用するとアプリケーションの動作を単純化することができる．

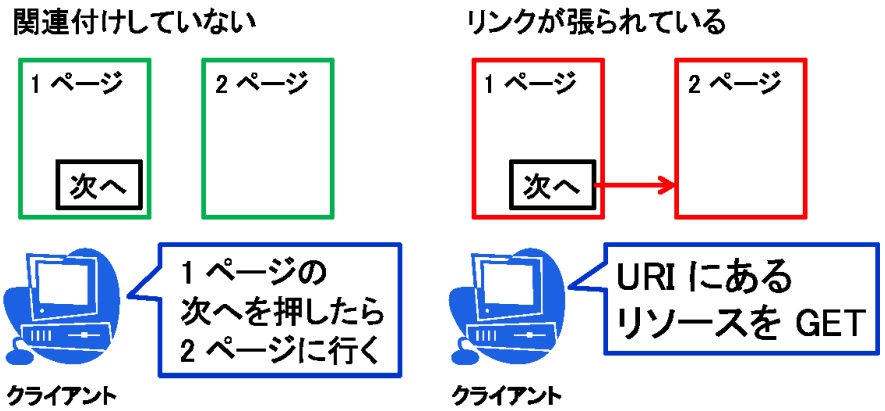


図 2.4: ハイパーリンクの使用

2.4.4 REST と Ajax

Ajax とは，非同期通信を行うための技術の総称である．ページ遷移を行うことなく，ページの一部のデータやインタフェースを更新することができる．これにより，ユーザビリティの向上や帯域の節約などが可能である．優れたユーザビリティは Web サービスにとって最も大切なものの一つであるため，多くの Web サービスが Ajax を利用している．しかし，2.2.3 でも述べたように Ajax には大きな欠点がある．それは，リソースの状態が変化しても，すべて同じ URI で管理される点である．その結果，アドレス可能性とステートレス性を失い，ユーザビリティやスケーラビリティを低下させる．これは Web アプリケーションにとって大きな問題である．そのため，Ajax を REST に適応させる必要がある．

例えば Google Maps は次のような方法を取っている．図 2.5 は実際の Google Maps の一部の画像である．Google Maps では図 2.5 の赤枠で囲まれている部分で，現在表示している地図の URI を取得することができる．つまり，ユーザはここで URI をコピーし，他のユーザと地図を共有することができる．Google Maps はこのようにアドレス可能性を付加させ，クライアントの状態の復帰を可能としている．その他の方法としては，Ajax アプリケーションにブラウザの戻るボタンや進むボタンを再実装することで，ステートレス性を取り戻す方法もある．つまり，重要なことは完全なステートレス性ではなく，ユーザ

が操作ミスなどで状態を戻す必要に迫られた際に，適切に状態を戻すようにできることである．

図 2.5: Google Maps 上での URI の取得

2.5 Publish and Subscribe

Publish and Subscribe (PubSub) とは，Publisher (発信者) と Subscriber (購読者) にクライアントを役割分担させるメッセージモデルである．クライアントを発信者と購読者に分けることで，サーバのスケーラビリティを向上させる．図 2.6 のように購読者はサーバに対して，購読のリクエストを送信する．リクエストが受理されると，購読者は論理チャンネルに登録される．この論理チャンネルのことをトピックを呼ぶ．図 2.7 のように発信者は各自のトピックにメッセージを送信し，購読者は登録しているトピックに届いた全てのメッセージを受信する．そのため，クライアントは相手のシステム，状態に関係なく通信を行うことが可能である．メッセージの送信もトピックに対して行われるので，相手のアドレスを知る必要がない．クライアントの増減も，トピックに追加・削除を行うのみなので容易である．　チャットなどのようにメッセージの送信・受信を共に行う場合は，Publisher かつ Subscriber となる．

2.6 Remote Procedure Call

Remote Procedure Call (RPC) [25] とは，別のネットワーク上にある手続きを呼び出すための技術である．TCP/IP のアプリケーション層に位置する．REST が主流となる前は多くの Web アプリケーションで使用されていた．XML にメソッド情報を含めることで，サーバ上で HTTP だけでは実現できない様々な手続きを呼び出すことができる．また，処理を他のサーバに行わせることで分散処理を可能とする．ただ，これはインタフェースの

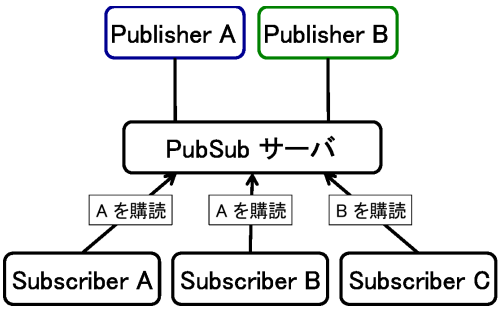


図 2.6: Subscriber の登録

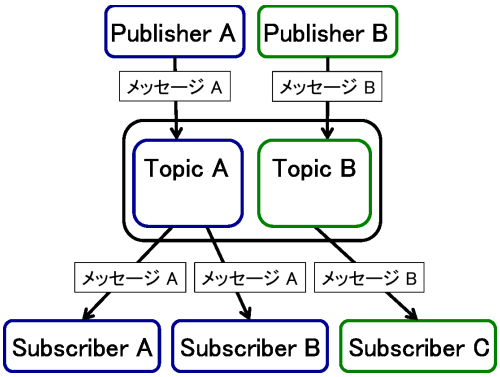


図 2.7: PubSub の通信

統一性をなくし，アプリケーションの互換性をなくすことを意味する．そのため，RPC スタイルの Web アプリケーションは RESTful と言えない．しかし，WebSocket はまだ統一インタフェースが定義されていないため，本研究で提案する統合モデルでは RPC スタイルを取る．

第3章　WebSocket

3.1　概要

従来の HTTP による擬似的なリアルタイム通信の課題を解決するために標準化されたのが WebSocket [1] である．2011 年 12 月に仕様が確定し，現在はいくつかの拡張について議論が行われている．WebSocket の最大の特徴は単一の TCP コネクションでステートフル通信を行っている点である．TCP を使うことで従来の技術に比べ，レイテンシ、オーバヘッド、実装の簡易性に優れたリアルタイム通信を行うことができる．現在は多くの言語でライブラリが開発され，すべての主要ブラウザ [26] [27] で使用することができる．しかし，現在公開されている多くのライブラリは，ハンドシェイクやデータ通信に関する最低限の機能を実装したものである．そのため，実際にサービスとして使用する場合はスケーラビリティやコネクションが切れた際の状態の復帰など，いくつかの WebSocket の課題を解決する必要がある．汎用的な WebSocket アプリケーションを開発するためには，WebSocket の課題を解決する，シンプルで拡張性の高い設計の提案が必要である．

3.2　従来のリアルタイム通信

WebSocket の開発以前はリアルタイム通信を実現するために HTTP 通信を使用していた．それが Polling や Long Polling といった技術である．Polling は定期的にリクエストを出すという単純な方法なため，リアルタイム性は低い．それに対して Long Polling はサーバと接続を維持し続けることで擬似的なプッシュ通信を行うため，リアルタイム性は高い．しかし，HTTP は本来ステートレスでリアルタイム通信に不向きな技術なため，いくつかの重要な課題が存在する．

3.2.1　Polling

Polling はリアルタイム性を向上させる最も単純な方法である．定期的かつ頻繁にリクエストをサーバに送ることで，サーバの更新を素早く反映させる．Ajax を使用した非同

期通信である．Polling の通信を図に表すと図 3.1 のようになる．矢印はクライアンとサーバの HTTP 通信を表している．このように，クライアントがサーバに対して，定期的に繰り返しリクエストを出すことでリアルタイム性を向上させる．

図 3.1 は 3 種類の状況の異なるリクエストを表している．一つ目のリクエストは Polling でリアルタイム性が保たれる通信の例，二つ目は無駄なリクエストの例，三つ目はレイテンシが発生するリクエストの例である．一つ目のリクエストは，サーバのデータ更新の直後にサーバに届いている．この場合，クライアントはサーバの更新をほとんどリアルタイムに取得することができる．次に二つ目のリクエストは，データ更新がある前にサーバに届いている．Polling は定期的にリクエストを送るステートレスな通信なので，サーバの更新のタイミングがわからない．そのため，このような無駄な通信を行う可能性がある．最後に三つ目のリクエストは，データの更新からしばらくたってからサーバに届いている．この場合はレイテンシが大きくなってしまい，リアルタイムな通信とは言えない．二つ目のリクエストと同様に頻繁に起こりうる．

つまり，Polling でリアルタイム性を保つことができるのは，リクエストが届く直前にサーバの更新があった場合のみである．多くの場合はレイテンシが発生し，もしくは無駄な通信となる．もしリアルタイム性を高めようとリクエストの間隔を小さくすると，サーバの負担が大きくなる上に，無駄な通信が増えてしまう．このように，Polling は単純で実装が容易だが，リアルタイム通信として課題が多い．

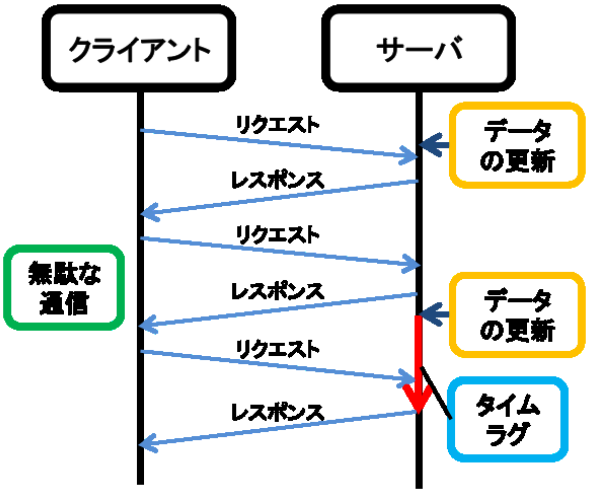


図 3.1: Polling 概略図

3.2.2 Long Polling

Long Polling は Polling のリアルタイム性の低さを解決した技術である．定期的にリクエストを送る部分は Polling と同様だが，リクエストを受けたあともすぐにレスポンスを返さずに接続を維持し続ける．そして，サーバにデータの更新があるとレスポンスを返す．こうすることで，HTTP 通信で半二重通信を可能にしている．図に表すと図 3.2 のようになる．この図は図 3.1 同様，3 種類の状況の異なるリクエストを表している．

一つ目のリクエストは，Long Polling で効果的な通信を行った場合の例である．Long Polling はすぐにレスポンスを返さないので，リクエスト後にデータ更新があった場合でもリアルタイムに反映することができる．二つ目のリクエストはタイムアウトになった例である．サーバに長時間更新が無い場合は，一定時間でタイムアウトとなる．よって，無駄な通信になってしまう．三つ目のリクエストはレイテンシのある通信の例である．前回のレスポンスと次のリクエストの間にサーバの更新があり，これによりタイムラグが発生している．このパターンはクライアントとサーバの距離が遠い場合や通信速度が遅い場合など，リクエストとレスポンスが届くのに時間がかかる場合に多く発生する．これは Long Polling が完全な二重通信ではなく，一つのレスポンスに対して一つのリクエストが必要なためである．このように，Long Polling は接続を維持することで擬似的なプッシュ通信を可能にし，リアルタイム性を向上させている．しかし，Polling ほど頻繁ではないが，無駄な通信やレイテンシも存在する．

さらに，Long Polling の最大の欠点はサーバへの負担が大きいという点である．これは HTTP でステートフルな通信を行っているためである．ステートフル通信はそもそもクライアントの接続を維持させるため，サーバのプロセスを占有してしまう．それに加え HTTP 通信はオーバヘッドが大きい通信技術でリソースを多く消費してしまう．また，Long Polling は通信に二重の HTTP コネクションを使用する．一つは通信を維持するための HTTP コネクション，もう一つはデータを通信するための HTTP コネクションである．このように，Long Polling は Polling と比べリアルタイム性は高いが，サーバへの負担が大きい．

3.3 WebSocket Protocol

WebSocket プロトコルは IETF で議論され，RFC6455 で定義されている．単一の TCP コネクションで通信を行うことで，全二重通信を行う．これによって，Long Polling の課

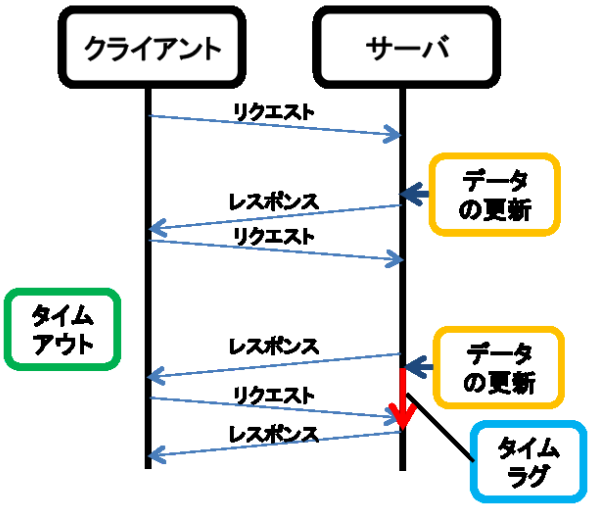


図 3.2: Long Polling 概略図

題であった，レイテンシやサーバへの負担，実装の複雑さを解決している．WebSocket 通信は，初めにクライアントとサーバで HTTP によるハンドシェイクを行う．ハンドシェイクが成功すると，TCP コネクションによる WebSocket 通信の開始である．データはフレームによって表現され，UTF-8 によって符号化されたテキストデータとバイナリデータが使用可能である．TCP コネクションは通信が終了するまで維持し続けられ，ステートフルな通信となる．そして，通信の終了の際にはステータスコードを含んだ Close フレームが転送される．

3.3.1 ハンドシェイク

ハンドシェイクは HTTP によって行われる．転送されるリクエストとレスポンスは図 3.3，図 3.4 のようになる．`Upgrade: websocket` によってサーバに WebSocket を使用することを通知する．その際，サーバが正しい WebSocket サーバであることを証明するために `Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==` を渡す．`Sec-WebSocket-Key` はサーバで GUID(Globally Unique Identifier) [28] を連結し，SHA-1 [29] でハッシュ化を行う．その後 Base64 [30] で符号化され，`Sec-WebSocket-Accept` ヘッダに渡される．この値が予期した値と違う場合，不正な WebSocket サーバとして，接続が拒否されるのである．また，リスト 3.3，リスト 3.4 にあるヘッダフィールドが欠けた場合もハンドシェイク失敗となる．サーバはハンドシェイクが成功した時のみステータスコード `101 Switching`

Protocols を返す.

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

図 3.3: ハンドシェイク時のリクエストの例

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

図 3.4: ハンドシェイク時のレスポンスの例

3.3.2 データフレーム

ハンドシェイク成功後は TCP コネクション上に図 3.5 のようなデータフレームが全二重通信で転送される．WebSocket はヘッダが小さく，最大でも 112 bit である．

WebSocket では，クライアントからサーバへのデータは全てマスクされる．そのため，マスク済みを表す `frame-masked` フィールドは必ず 1 の値を取り，それ以外の値の場合は WebSocket 接続が切断される．マスキングキーはクライアントが 32 bit の値でランダムに生成し，その値を `frame-masking-key` フィールドにセットする．安全なマスキングキーの生成については RFC4086 [31] で定義されている．

また，WebSocket は拡張性が高くなるよう設計されていて，拡張のためのフレームが用意されている．拡張はハンドシェイクの際に `Sec-WebSocket-Extensions` ヘッダーフィールドによって交渉を行う．交渉が成立すると，`frame-rsv` フィールドに 0 以外の値をセッ

トする．そして，拡張データを frame-payload-data フィールドに含む．拡張がないにも関わらず `frame-rsv` フィールドが 0 以外の値を取っている場合，WebSocket 接続は切断される．

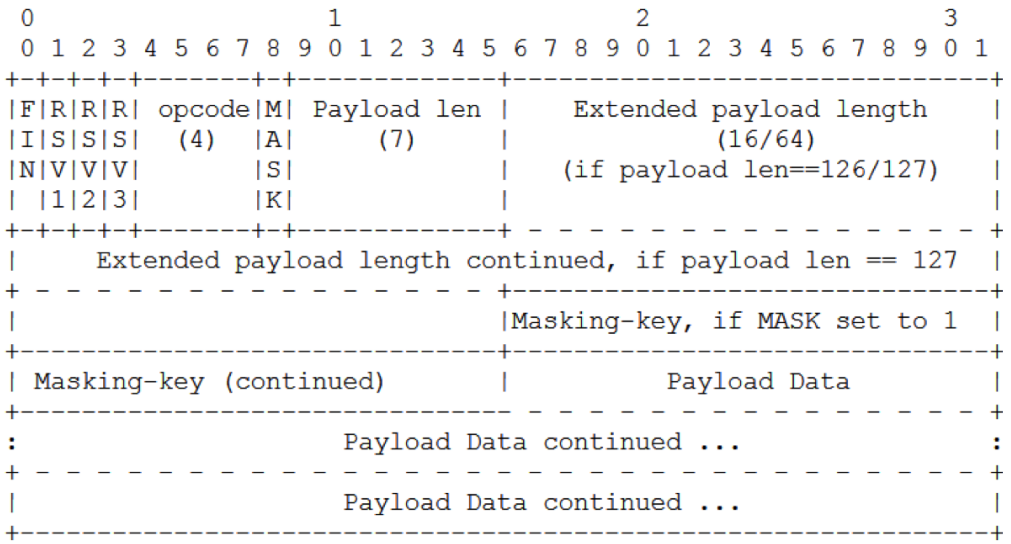


図 3.5: データフレーム [1]

3.3.3 WebSocket の有用性

WebSocket はレイテンシが低い，オーバヘッドが小さい，実装が簡易的という理由から，従来のリアルタイム通信に比べ有用性が高い．これらの利点を Long Polling と WebSocket を比較して示す．

Communicating and Displaying Real-Time Data with WebSocket [32] は Victoria Pimentel らが行ったレイテンシを比較した実験である．この実験では 3 つの時間帯で，カナダの大学からカナダ，ベネズエラ，スウェーデン，日本の大学と Polling，Long Polling，WebSocket の 3 種類の通信を行いレイテンシを計測した．その結果，ほとんどの地域，時間帯で WebSocket のレイテンシが最も低くなった．ベネズエラの 12:00 台，スウェーデンの 10:00 台のみ Long Polling の方がレイテンシがわずかばかり低くなったが，誤差の範囲内である．Pollingh は常に WebSocket よりもレイテンシが高かった．この実験から WebSocket が最もリアルタイム性が高いといえる．

次にヘッダを比較する．HTTP のヘッダは通常 300 から 400 バイト程度である．それに

対して WebSocket は最大でも 14 バイトである．つまり WebSocket が最大のヘッダ長をとった場合でも，HTTP は 20 倍以上のヘッダ長となる．リアルタイム通信では，小さいデータを頻繁に通信することが想定されるので，毎回 20 倍以上のヘッダが余計にかかるのは大きな無駄となる．よって，WebSocket はオーバヘッドが小さいといえる．

そして最後に実装についてである．WebSocket は単一の TCP コネクションで通信を行う．それに対して Long Polling は，コネクションの維持とデータ通信のために HTTP コネクションを二重に使用する．また，HTTP でステートフル通信を行うことは本来の使い方ではない．そのため，Long Polling は実装が複雑性になってしまう．以上の理由から WebSocket はリアルタイム Web に最も適した通信である．

3.4 拡張

3.4.1 Compression Extensions for WebSocket

WebSocket で圧縮データを転送するための拡張プロトコルである．データ圧縮を行うことでより大きいデータの通信に対応することができる．現在 IETF によって議論中で，最新のバージョンは 2014 年 1 月 22 日に公開された `permessage-compression-17` [33] である．

データの圧縮は DEFLATE アルゴリズム [34] によって行う．DEFLATE は可逆データ圧縮アルゴリズムで，圧縮率は低いが，圧縮速度が早いことが特徴である．HTTP 通信のデータ圧縮や ZIP，PNG など広く使われている技術である．しかし，DEFLATE は SSL 通信と同時に使用すると暗号を解読され，通信を傍受されてしまうセキュリティ問題がある．この脆弱性は Juliano Rizzo と Duong Thai によって指摘され、それに対する攻撃は CRIME (Compression Ratio Info-Leak Made Easy) 攻撃 [35] と呼ばれる．そのため圧縮を行う場合，SSL 通信は使用できない．

拡張を使用するにはサーバに許可を得る必要がある．この許可を得る手続きを交渉と呼び，ハンドシェイクの際に行われる．クライアントは `Sec-WebSocket-Extensions` ヘッダに値 `permessage-deflate` を入れて，サーバにリクエストを出す．交渉が成立した場合，WebSocket コネクションを確立し，`frame-rsv` フィールドの値が 100 のデータフレームを転送する．

3.4.2 A Multiplexing Extension for WebSockets

WebSocket で多重化を行うための拡張プロトコルである．多重化を行うことで，クライアントに対して同時に複数の WebSocket サービスの利用を可能にする．また，サーバのスケーラビリティを向上にも利用できる．現在 IETF によって議論中で，最新のバージョンは 2013 月 7 月 2 日に公開された `websocket-multiplexing-11` [36] である．

多重化は TCP コネクションを論理チャンネルによって，仮想的に分割することで実現する．それぞれのチャンネルには ID が与えられ，サーバはチャンネル ID によって送信元の論理チャンネルを判別する．交渉はハンドシェイクの際に行われる．`Sec-WebSocket-Extensions` ヘッダの値は `mux` である．多重化コネクションが接続されると，チャンネル ID などがフレームに付加される．0 のチャンネル ID を持つ論理チャンネルは，multiplex control block をサーバと通信する．multiplex control block は，チャンネルの追加や削除などに使用される．

3.5 WebSocket フレームワーク

WebSocket を扱いやすくするためのフレームワークやプロトコルがいくつか公開されている．WAMP はメッセージング機能を強化した RPC スタイルの通信を行うプロトコル，Realtime Framework は WebSocket の実装をサポートするためのフレームワークである．

3.5.1 WAMP

概要

WAMP(The WebSocket Application Messaging Protocol) [37] はハイレベルなメッセージングのための WebSocket 拡張プロトコルである．Tavendo 社によって策定され，公開されている．最低限のメッセージング機能しか持たない WebSocket に，PubSub と RPC の機能を追加し，より高機能なメッセージングを可能にする．通信データには JSON が使用される．多くの言語でサーバ・クライアント向けのライブラリが実装されており，Tavendo 社も Autobahn というライブラリを公開している．

プロトコル

WAMP は WebSocket コネクション上で，9 つのタイプのメッセージを通信する．表 3.1 はメッセージタイプ，タイプ ID，通信の向き，メッセージの用途をまとめたものである．

メッセージは JSON によって表記され，全てのメッセージでタイプ ID，URI が指定される．表 3.1 からわかるように，メッセージは補助，RPC，PubSub の 3 通りの用途がある．

WAMP ではコネクションを張ると，最初に必ず WELCOM メッセージが送信される．メッセージにはセッション ID，使用されている WAMP のバージョン ID，プラットフォームなどサーバの情報が含まれる．PREFIX メッセージは CURIE (Compact URI Expression)[38] を使用する場合に送信される．URI と prefix をサーバに送る．

RPC のためには 3 つのメッセージが用意されている．CALL メッセージをサーバに送信し，それに対してサーバは CALLRESULT か CALLERROR を返す．全ての RPC メッセージには callID が付けられ，callID によってメッセージを区別する．CALLERROR には必ずエラー詳細が書かれ，エラーの内容をクライアントに知らせる必要がある．

PubSub のためには 4 つのメッセージが用意されている．購読と非購読のための，SUBSCRIBE と UNSUBSCRIBE メッセージ，パブリッシャーがトピックを発信するための PUBLISH メッセージ，PUBLISH メッセージをサブスクライバーに伝えるための EVENT メッセージの 4 種類である．全ての PubSub メッセージには topicURI が付けられ，トピックを区別する．WAMP バージョン 1 では，PubSub のエラーをフィードバックするための，メッセージは用意されていない．そのため，エラーがあった場合，勝手にサーバに破棄される．

表 3.1: WAMP で使用されるメッセージ

メッセージ	タイプ ID	方向	用途
WELCOM	0	Server-to-client	Auxiliary
PREFIX	1	Client-to-server	Auxiliary
CALL	2	Client-to-server	RPC
CALLRESULT	3	Server-to-client	RPC
CALLERROR	4	Server-to-client	RPC
SUBSCRIBE	5	Client-to-server	PubSub
UNSUBSCRIBE	6	Client-to-server	PubSub
PUBLISH	7	Client-to-server	PubSub
EVENT	8	Server-to-client	PubSub

3.5.2 Realtime Framework

Realtime Framework [39] とは IBT 社が提供する WebSocket を扱うためのフレームワークである．抽象化レイヤーの ORTC (Open Realtime Connectivity) と開発レイヤーの xRTML (Extensible Realtime Multiplatform Language) [40] の二つのレイヤーから構成される．

抽象化レイヤーである ORTC はサーバとリアルタイム通信を担当する．スケーラビリティの向上のために，コネクションの多重化，サーバの並列化を可能にしている．また，チャンネルを使用し，個対多や多対多の通信を行う．チャンネルベースでアクセス権限を付加も可能である．

xRTML はリアルタイム通信を実現するためのマークアップ言語である．xRTML で実装したものを，ORTC サーバが処理する．様々なクラス [41] が用意されていて，WebSocket を意識することなく通信が実装可能である．サーバ側は Ruby，PHP，Java など多くのプログラミング言語が使用できる．クライアント側は JavaScript で実装する．

3.6 WebSocket ライブラリ

3.6.1 概要

現在, 多くの WebSocket ライブラリがオープンソースで公開されている．EM-WebSocket [42] は Ruby 用の WebSocket ライブラリで最も人気のあるライブラリである．機能がシンプルで，WebSocket を実現するための最低限の機能を備えている．Socket.IO [43] は Node.js で提供されている最も有名な WebSocket ライブラリの一つである．他のプログラミング言語からも Socket.IO を扱うためのライブラリが公開されている．高機能でユーザをサポートするため機能がいくつか提供されている．Autobahn [44] は WAMP を実装したライブラリである．WAMP を実現するためのライブラリはいくつかあるが，Autobahn は WAMP を規格している Tavendo 社が開発したライブラリである．サーバ向けに Python 用ライブラリ，クライアント向けに Python，JavaScript，Java (for Android) 用ライブラリを公開している．また，300 以上のテストも用意されている．上記のライブラリは全てオープンソースである．

3.6.2 EM-WebSocket

EM-WebSocket は Ruby 用 WebSocket ライブラリである．Ruby のライブラリである EventMachine を使用して，Reactor パターンを実現している．WebSocket を実装するための最低限の機能しか持っていない．ハンドシェイクなどの通信部分，ステータスコードのコールバック，TSL 通信，デバッグ機能などを提供する．機能が少ない分シンプルで可読性は高いが，再接続やスケーラビリティなどの問題は利用者が個別に対応する必要がある．

3.6.3 Socket.IO

Socket.IO は Node.js 用 WebSocket ライブラリである．EM-WebSocket と比べると高機能である．ハンドシェイクや通信機能以外にも，タイムアウトやバッファを使用したクライアントの再接続，名前空間を使用したコネクションの多重化などの機能を持つ．多様なクライアントに対応するために，ソケットをオープンする際に対応可能な通信を判別して自動で最適な通信に切り替える．例えば WebSocket が使えないクライアントには Ajax や Comet に切り替える．

また，Socket.IO は room という概念を採用していて，room によって個別やグループ，全体との通信の切り替えを容易にし，多対多や個対多の通信を行うことが可能である．他にも認証のための機能などが提供されている．

3.6.4 Autobahn

Autobahn は WAMP をサポートするオープンソースのライブラリである．Python, JavaScript，Java で実装されている．特徴としては非同期処理や RPC と PubSub メッセージパターンの提供，自動再接続，TLS のサポートが上げられる．また，テストが用意されているため，WAMP に準拠しているかを簡単に確かめることが出来る．

3.7 WebSocket の課題

3.7.1 概要

WebSocket はステートフル通信であるため，Web サービスで使用する際にいくつかの課題が存在する．特に代表的な課題を以下に挙げる．これらの WebSocket の課題に対応していくことが本統合モデルの目的であり，今後の課題である．

3.7.2 スケーラビリティの問題

ステートフル通信はステートレス通信に比べ，スケーラビリティが低い．これはクライアントによるサーバの占有と依存が原因である．クライアントは一度サーバと通信を開始すると切断するまで接続を維持し続ける．そのため，サーバはクライアントにプロセスの一部を与え続けなければならない．

また，ステートフル通信はクライアントの状態をサーバが管理するので，特定のサーバとしか通信が行えない．それに対してステートレス通信は各リクエストが独立していて，クライアントは全リクエストに自分の状態を含めるので，リクエストごとにサーバを変えることができる．そのため，ステートレス通信は単純にサーバを増設するだけで，スケーラビリティを向上させることができる．大規模なサービスを運用する場合，ステートフル通信でどのように負荷分散を行うかが課題となる．

3.7.3 コネクションの管理

ステートフル通信の場合は，接続を維持し続けなければ通信が行えないため，コネクションが確実に張られていることを確認する必要がある．そして，コネクションが切れた場合はクライアントやサーバに通知する仕組みが必要である．また，コネクションの管理として重要な問題として，非アクティブユーザの検出がある．既にサービスの利用を終了しているユーザは，サーバのスケーラビリティのために切断するべきである．必要なコネクションや正当なコネクションを判断する方法，そして，それをサーバやクライアントに通知する仕組みが必要である．

3.7.4 状態の復帰

ステートフル通信ではクライアントの状態をサーバが管理する．そのため状態の復帰が課題となる．例えば，ステートフル通信はブラウザの戻るボタンを使用することができない．これは，ステートフル通信はユーザが特定の順序を踏む必要があるためである．A の次は B，その次は C，という依存関係があるリソースを想定すると，C から B に戻る場合は必ず A のリソースから通信をやり直さなければならない．それに対して，ステートレス通信は各リクエストが独立しているので，手順に関係なく戻ることができる．

また，クライアントの再接続でも状態の復帰は課題となる．上記と同様の理由でステートフル通信は，再接続の際に最初からリソースをたどる必要がある．そのため，突然の切

断の際に状態の復帰が困難である．例えば，再接続の際にクライアントを識別して最新の情報をプッシュする方法を取るとすると，サーバが切断後もしばらくクライアントの状態を維持していなければならない．これはサーバに大きな負担である．

第4章　RESTとWebSocketの統合モデルの提案

4.1　概要

本研究はRESTとWebSocketアプリケーションの統合モデルの提案を行い，いくつかのアプリケーションで統合モデルの検証を行った．統合モデルの目的は，WebSocketアプリケーションをRESTに対応させるための設計方法の提案である．現在RESTはWebの主流であり，ほとんどのWebアプリケーションが考慮するべき設計スタイルである．RESTの利点については第2章で解説している．そして，これらの利点はWebSocketの課題解決にも有効である．しかし，WebSocketでRESTfulを実現するためはいくつかの問題がある．その一つがWebSocketのステートフル性である．本研究ではこの問題を解決することでRESTの利点をWebSocketアプリケーションに継承させる．また，本研究の統合モデルは特にWebSocketアプリケーションのユーザビリティを向上させる．

4.2　WebSocketアプリケーションの現状

現在，すべての主要ブラウザでWebSocketによる通信が可能である．それに伴いWebSocketを使ったアプリケーションも登場している．例えば株チャートやブラウザ上で複数人と共同作業をするアプリケーションなどである．これらはデータ更新頻度が高く，リアルタイム性が重要となるアプリケーションである．特に複数人によるブラウザ上での共同作業は更新タイミングが一定ではないためPollingよりもWebSocketの方が有効である．このようにリアルタイムWebアプリケーションに効果的なWebSocketだが，これらのアプリケーションはいくつかの解決すべき課題がある．その一つがRESTへの対応である．

現在，多くのWebSocketアプリケーションはRESTに対応していない．つまり，Webアプリケーションの最も優れた特徴であるアドレス可能性を持っていない．これはユーザビリティにとって大きな問題である．そのため，Webアプリケーションの主要な通信方法として発展していくためには，WebSocketアプリケーションのためのリソース指向アーキ

テクチャの確立が必要である．本統合モデルは WebSocket アプリケーションの REST への対応を行い，この問題を解決する．

4.3 統合モデル

4.3.1 定義

本研究は REST と WebSocket アプリケーションとの統合モデルを提案する．しかし，統合モデルとは完全に RESTful であるという意味ではない．本研究では RESTful の定義に ROA の基準を採用する．そのため，RESTful アプリケーションとするには統一インタフェース，アドレス可能性，ステートレス性，接続性が必要である．しかし，WebSocket はステートフル性が特徴の通信であるため，完全にステートレスにすることはできない．そこで，本統合モデルは WebSocket の長所を残し，考慮すべき REST の特徴を統合したものとなっている．また，多くの議論が必要である統一インタフェースは本統合モデルでは扱わない．

4.3.2 目的

本統合モデルの目的は，WebSocket アプリケーションを REST に適応させるための一つの解決案を提示することである．またこの提案により，個々のアプリケーションを関連なく設計することを回避する．そして，REST に適応することでリソース指向である現在の Web に合ったアプリケーションの開発が可能となる．これは Web の重要な特徴である URI によるリソースの参照を可能にし，ユーザビリティを向上させる．また，WebSocket にステートレス性を与え，WebSocket の課題である状態の復帰を解決する．他にもいくつかの課題に対して有効である．

4.4 提案

4.4.1 概要

本研究で提案する統合モデルは WebSocket アプリケーションを REST に適応するための設計方法である．WebSocket アプリケーションにアドレス可能性を付加させ，ステートレス性と接続性を向上させる．また，WebSocket をリソース指向にするためのリソースの扱い方法の提案も行う．本統合モデルにより WebSocket アプリケーションのユーザビ

リティを向上させることが可能である．ただし，本統合モデルは完全な RESTful アプリケーションを提供するものではない．RESTful にとって重要である統一インタフェースを扱っていないためである．統一インタフェースは様々な面から議論が必要なルールであり，WebSocket の今後の課題である．

4.4.2 アドレス可能性の付加

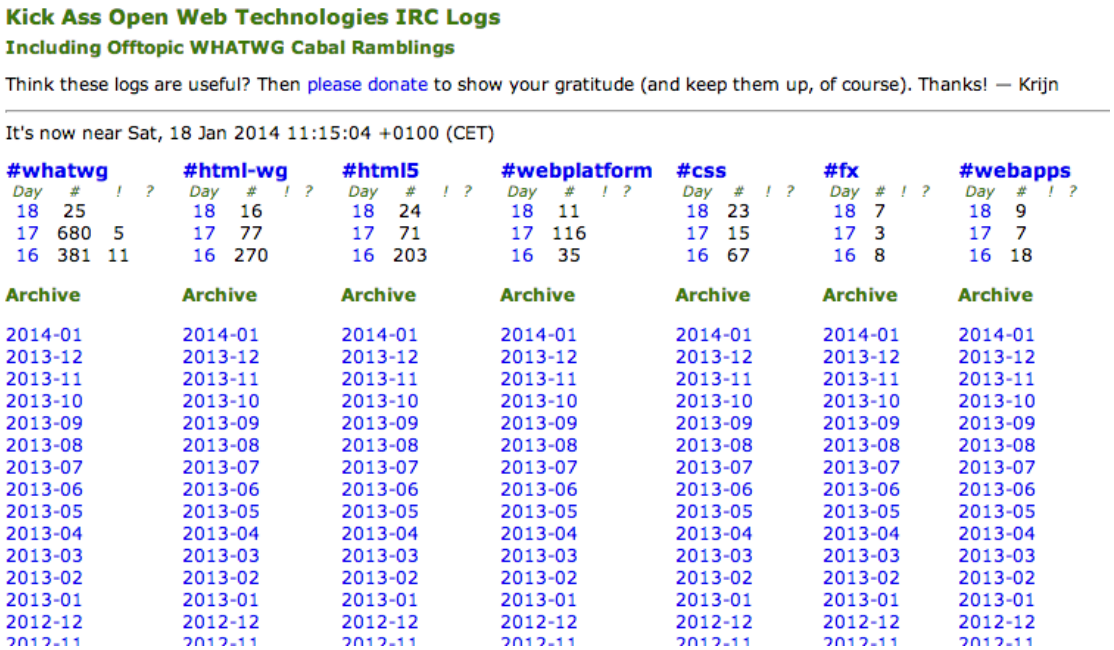
WebSocket が RESTful でない最大の理由は WebSocket のステートフル性である．WebSocket はステートフルに通信を行い，非同期にデータを更新する．その結果，WebSocket アプリケーションは複数のリソース状態を一つの URI で管理し，アドレス可能性や接続性，ステートレス性を失う．そこで REST と WebSocket を統合するために，WebSocket アプリケーションにアドレス可能性を付加させる．アドレス可能性の付加は統合モデルで最も大切な要素である．アドレス可能性によって，接続性やステートレス性を向上させることが可能である．

WebSocket アプリケーションにアドレス可能性を付加させるためには，一つの URI で複数のリソース状態を管理するのをやめる必要がある．そこで，リソースの状態が変化するたびにリソースに URI を与える．これにより URI で操作するリソース状態を指定することが可能になり，各リソースはアドレス可能となる．このとき URI はクエリの部分を変化させる．URI はホスト，パス，クエリ，フラグメントによってリソースを一意に識別するが，ホストやパスはアプリケーションの位置を表しているのでこの場合は変更されない．また，フラグメントはリソース内の特定の場所を指定する際に使われるため，フラグメントを利用するのも適当でない．クエリはサーバにパラメータを渡すのに使われる文字列で，リソースのデータに合わせて指定することが可能なので今回の問題に適している．そのため，リソースのデータの変化に合わせて，呼び出しに必要な値をクエリに追加する．例えばリソースが時間で指定可能ならば `?time=` とし，イコール以降の値を変更する．データの種類が増えた場合も `"&"` で変数を増やすことで対応可能である．クエリをリソースの状態に合わせて変化させることにより一意にリソースを識別し，状態の復帰や URI による参照が可能となる．

4.4.3 接続性の向上

WebSocket アプリケーションにアドレス可能性を付加することで，接続性を向上させることができる．各リソースの状態は URI と結びついているため，ハイパーリンクによる

相互接続が可能である．これは，各リソースの状態の意味を明確にし，体系化や理解に有効である．また，アプリケーションの操作性を単純化させ，ユーザビリティを向上させる．接続性のある Web サイトの例として IRC Logs (`http://krijnhoetmer.nl/irc-logs/`) というサイトがある．IRC とはサーバを介してクライアント同士でチャットを行うシステムである．このサイトは IRC でのチャットの会話ログを残して，Web 上で観覧できるようになっている．図 4.1 は IRC Logs のトップページである．青文字の部分はすべてリンクとなっていて，"#" の部分はチャットの議題，数字部分はチャットのログの年と月を表している．ログを見たい年月のリンクをクリックすると，図 4.2 のような日付けのページに遷移する．さらに図 4.2 の日付けのリンクをクリックすると，図 4.3 のように会話ログが表示される．このように，接続性によってユーザのリソースの操作を容易にし，リソースの意味を簡単に理解することが可能となる．IRC Logs がこのように各リソースを接続することができるのは，すべてのリソースを日時ごとに URI で管理しているためである．アドレス可能性によって接続性を向上させることが可能となり，接続性の向上がユーザビリティやアプリケーションの簡易性を向上させる．



Kick Ass Open Web Technologies IRC Logs

Including Offtopic WHATWG Cabal Ramblings

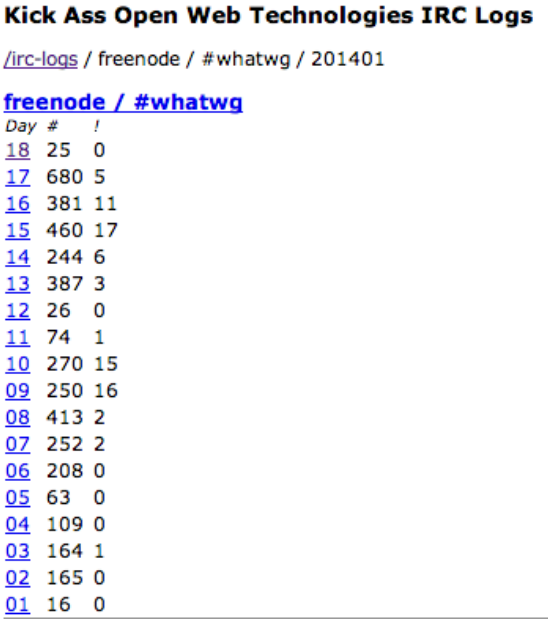
Think these logs are useful? Then please donate to show your gratitude (and keep them up, of course). Thanks! — Krijn

It's now near Sat, 18 Jan 2014 11:15:04 +0100 (CET)

#whatwg				#html-wg				#html5				#webplatform				#css				#fx				#webapps			
Day	#	!	?	Day	#	!	?	Day	#	!	?	Day	#	!	?	Day	#	!	?	Day	#	!	?	Day	#	!	?
18	25			18	16			18	24			18	11			18	23			18	7			18	9		
17	680	5		17	77			17	71			17	116			17	15			17	3			17	7		
16	381	11		16	270			16	203			16	35			16	67			16	8			16	18		

Archive	Archive	Archive	Archive	Archive	Archive	Archive
2014-01	2014-01	2014-01	2014-01	2014-01	2014-01	2014-01
2013-12	2013-12	2013-12	2013-12	2013-12	2013-12	2013-12
2013-11	2013-11	2013-11	2013-11	2013-11	2013-11	2013-11
2013-10	2013-10	2013-10	2013-10	2013-10	2013-10	2013-10
2013-09	2013-09	2013-09	2013-09	2013-09	2013-09	2013-09
2013-08	2013-08	2013-08	2013-08	2013-08	2013-08	2013-08
2013-07	2013-07	2013-07	2013-07	2013-07	2013-07	2013-07
2013-06	2013-06	2013-06	2013-06	2013-06	2013-06	2013-06
2013-05	2013-05	2013-05	2013-05	2013-05	2013-05	2013-05
2013-04	2013-04	2013-04	2013-04	2013-04	2013-04	2013-04
2013-03	2013-03	2013-03	2013-03	2013-03	2013-03	2013-03
2013-02	2013-02	2013-02	2013-02	2013-02	2013-02	2013-02
2013-01	2013-01	2013-01	2013-01	2013-01	2013-01	2013-01
2012-12	2012-12	2012-12	2012-12	2012-12	2012-12	2012-12

図 4.1: IRC Logs のトップページ



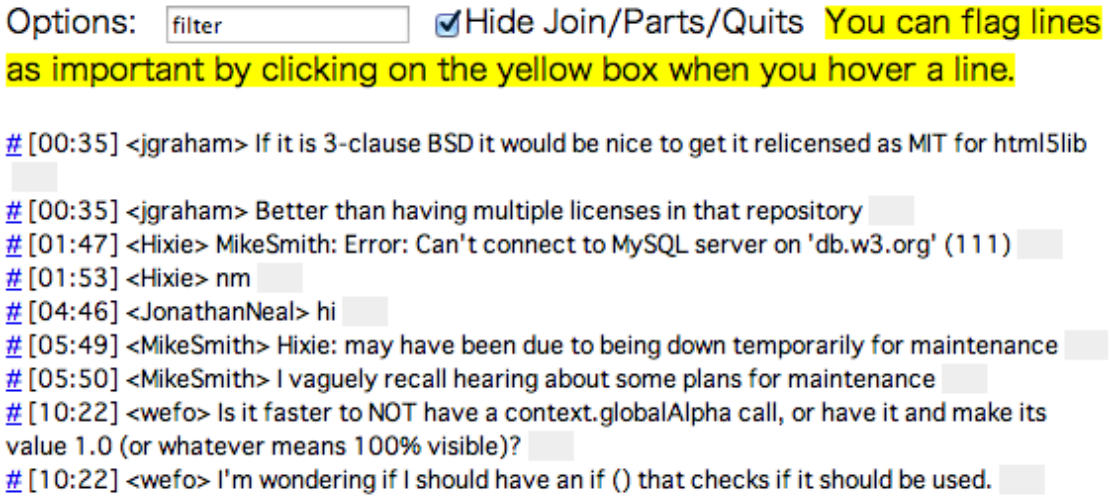
Kick Ass Open Web Technologies IRC Logs

/irc-logs / freenode / #whatwg / 201401

freenode / #whatwg

Day	#	!
18	25	0
17	680	5
16	381	11
15	460	17
14	244	6
13	387	3
12	26	0
11	74	1
10	270	15
09	250	16
08	413	2
07	252	2
06	208	0
05	63	0
04	109	0
03	164	1
02	165	0
01	16	0

図 4.2: IRC Logs の日付けページ



Options: filter Hide Join/Parts/Quits You can flag lines as important by clicking on the yellow box when you hover a line.

[00:35] <jgraham> If it is 3-clause BSD it would be nice to get it relicensed as MIT for html5lib

[00:35] <jgraham> Better than having multiple licenses in that repository
[01:47] <Hixie> MikeSmith: Error: Can't connect to MySQL server on 'db.w3.org' (111)
[01:53] <Hixie> nm
[04:46] <JonathanNeal> hi
[05:49] <MikeSmith> Hixie: may have been due to being down temporarily for maintenance
[05:50] <MikeSmith> I vaguely recall hearing about some plans for maintenance
[10:22] <wefo> Is it faster to NOT have a context.globalAlpha call, or have it and make its value 1.0 (or whatever means 100% visible)?
[10:22] <wefo> I'm wondering if I should have an if () that checks if it should be used.

図 4.3: IRC Logs の会話ログ

4.4.4　ステートレス性の向上

WebSocket はステートフル通信である．ステートフル通信はサーバがクライアントの状態を維持する．そのため，クライアントは特定のサーバに依存することになる．また，ステートフル通信はブックマークやブラウザの戻るボタンなどを無効にする．これは，ステートフル通信が過去のリクエストに依存しているためである．しかし，本統合モデルはアドレス可能性と接続性によって，これらの機能を再び利用可能にする．

アドレス可能ということは，URI によって取得するリソースを指定できるという意味である．これは，クライアントの URI による状態の復帰を可能とする．この特性を利用し，WebSocket アプリケーションにステートレス性を取り戻す．図 4.4 はステートフル通信と URI によるリソースの状態遷移を表した図である．赤い矢印はステートフル通信のリソースの状態遷移，青い矢印は URI によるリソースの状態遷移である．A，B，C はリソースの状態を表し，順序に意味があるとする．例えばオセロゲームなら，A は一手目，B は二手目，C は三手目を表す．図 4.4 の赤い矢印のように，ステートフル通信は各リクエストが独立していないため，A から C に移動するためには，A，B，C という道順をたどる必要がある．そのため，C から A に戻ることや，A と B を飛ばして C に行くことはできない．しかし，各リソースに URI が関連づけられていれば，URI を指定することで A，B，C の好きなリソースに遷移することができる．図 4.4 の青い矢印を見ると，とてもステートレス通信に近い挙動となっている．つまり，アドレス可能性で一部ステートレス性を取り戻すことができるのである．これにより，ブラウザの戻る機能やブックマークなどが使用可能である．

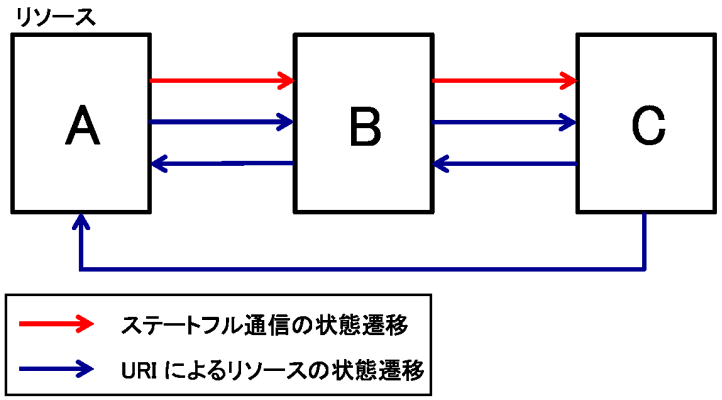


図 4.4: ステートフル通信と URI によるリソースの状態遷移

4.4.5 統一インタフェース

WebSocket で定義されているメソッドはデータ通信のための send メソッドとコネクション切断のための close メソッドである．そのため，実際にアプリケーションを実装する場合は自分で必要なメソッドを定義し，それを send メソッドを使って呼び出すという RPC スタイルを取ることになる．これは REST の統一インタフェースに反する．統一インタフェースはアプリケーションの孤立化をなくし，汎用性，拡張性を高めるためにとても重要である．しかし，本研究の統合モデルでは統一インタフェースについて言及しない．統一インタフェースは，多くのアプリケーションの開発から適したメソッドを提案し，それに対して様々な立場から議論をする必要がある．HTTP も同様の手順で様々な議論が行われている．大切なことは多くの人達が同じインタフェースを同じ方法で使用することである．

4.4.6 リソース

アドレス可能性を持たせる際に問題となるのがリソースの扱いである．リソースとは，実在する物体から正義といった概念まで Web 上に存在するすべてのものを指す．例えば新鮮なリンゴを放置していると時間が経つと痛んでいき，やがて完全に腐ってしまう．このとき"新鮮なリンゴ"，"一部痛んだリンゴ" といったリンゴの変化一つ一つもリソースである．つまり，時間に対して連続的に変化するリソースは無限に増加してしまう可能性がある．この際，どこまでリソースとして扱うかがアプリケーションの重要な課題となる．ここでは具体的な例を用いて本研究の統合モデルのリソースの扱いを説明する．

離散的な変化

例えばボードゲームやチャットなど，ある時間に瞬間的に違う状態に変化するアプリケーションを考える．これは連続的な変化をするリソースに比べ，リソース状態が爆発的には増えないため扱いが容易である．例えばチェスや囲碁なら一手ごとをリソースとして扱い，チャットならユーザの発言ごとをリソースとして扱えばよい．これにより，チェスならば一手ごとの履歴の参照，チャットならば図 4.3 のようにユーザの発言をログとして残すことができる．このように，離散的な変化をするアプリケーションはすべての変化をリソースとして扱えばよい．

しかし，離散的な変化をするアプリケーションでも状態数が膨大になることはある．例えば発言数が多く，多数のユーザが接続するチャットアプリケーションの場合，無限では

ないが URI の更新が相当な頻度で行われてしまう．これはユーザにとって不利益である．これに対する解決法は 2 通りある．リソースの扱い方を変えるか，URI の更新タイミングを変える方法である．例えばリソースを減らしたければ，リソースを日や時間単位で管理すれば良い．これで，管理するリソース状態を大幅に減らすことができる．あるいはリソースの扱いを変えたくない場合，リソースの状態は発言ごとに管理するが，ブラウザの URI の更新は一定時間，または発言数ごとに行えば良い．これによって，ユーザは適切な場面に容易に戻ることができるであろう．発言数ごとはページの概念に近い．ここで重要なことはチャットアプリケーションにおける発言の重要度である．各発言が重要だと考える場合は発言すべてをリソースとして扱い，そうでない場合は一定時間や日単位で扱えば良い．例えば Twitter (`https://twitter.com/`) はすべての発言がアドレス可能である．そうすることでリソース数は膨大になるが，Twitter は他のユーザの発言を参照することが可能である．つまり，そのリソースが参照に値するかどうかが大きな判断基準となる．

連続的な変化

例えば前例のリンゴの変化のように，ある期間で連続的に状態が変化するアプリケーションを考える．これは扱い方によって膨大なリソース数となってしまうが，更新頻度の高い離散的な変化と同様にリソースの重要度によって決定すればよい．重要な状態のみを取り出し，リソース化する．リンゴの変化ならば，目に見てわかる変化や，日にちごと，一定時間ごとなどである．そのアプリケーションがリンゴの何を重要としているかによって決まる．

他の例として Google Map の変化を考える．Google Map は中心点の座標を使ってリソースを区別している．そのため，すべての移動に対して URI を更新してしまうと，少しの移動でかなりの更新数となる．これでは，ユーザは戻るボタンで任意の場所に戻ることは困難である．そこで三通りの方法で URI を更新することでこの問題を解決してみる．意味や単位などのリソースの内容に適応したもの，特殊なイベント，ユーザの動作による URI の更新である．

一つ目の意味や単位による方法は，地図の地名や時間経過による URI 更新である．地名による更新とは，縮尺に合わせて，国，県や市など区域の境界線を超えた場合に更新することである．リンゴの観察ではこの方法で十分である．なぜならば，リンゴの観察では多くの状態変化が重要となるためである．しかし，地図の場合あまり良いアプローチではない．ユーザは自分の目的地など一部の地点には関心を持っているが，自分が県を超えた

瞬間に戻りたいわけではない．一定時間による更新も同様で，特定の時間に自分がどの地点を見ていたかを気にするユーザは少ない．

二つ目の特殊なイベントによる方法は，ボタンやピンを立てるなどのユーザによる操作で行う URI 更新である．これは確実に自分が気になった地点に戻ることができるため適している．しかし，自動で URI の更新がおこなわれないため，ユーザに負担をかけない簡易的な方法で実現する必要がある．

最後にユーザの動作とは，ユーザの動作を分析し，URI を更新する方法である．Google Map ならばユーザは目的地を探している間は地図を動かし，目的地を見つけた場合は動きを止めるであろう．そのため，一定時間動きがない場合などに URI を更新する．動作が止まった地点のすべてがユーザの目的地とは限らないが，自動で URI の更新が可能であり，ユーザに負担をかけずに無制限な URI の更新を防ぐことができる．これによってユーザは，数回の戻る操作で過去に重要と見なした状態に戻ることが可能である．

リソースの扱いと URI の更新

離散的，連続的なリソース状態の変化で共通するリソースの考え方は，そのリソースが重要かどうかである．つまり，参照する必要のあるリソースのみをリソースとして扱う．しかし，この基準を適応したリソースの場合も，状態の変化するタイミングが早いアプリケーションの場合 URI の更新が問題になる．その場合はすべてのリソースの変化に対して URI の更新を行わずに，ユーザが適切に過去の状態に戻れるよう一部のリソースに対して URI を更新するべきである．リソースの意味や単位，ユーザによる特殊なイベント，ユーザの動作に合わせた URI の更新である．URI の更新タイミングを考慮することで，ユーザビリティを損なうことなくアドレス可能性の付加が可能である．

4.5 提案の有用性

本節では，REST 以外の Web アプリケーションの設計方法の考察と本統合モデルの利点を述べることで本統合モデルの有用性を示す．

4.5.1 REST 以外のアーキテクチャスタイル

本研究では REST の利点を述べ，WebSocket アプリケーションの REST への対応の重要性を示した．REST の拡張性，汎用性，簡易性の高さは強力な利点であり，本統合モデ

ルはリアルタイム Web の発展に有益な提案である．しかし，すべての Web アプリケーションが REST に適しているとは限らない．そこで REST 以外の Web アプリケーションの設計方法である SOAP[45]，WS-* について考察する．

例えばインターネットバンキングの口座振り込みサービスを考える．ユーザが A の口座から B の口座にお金を振り込む場合，アプリケーションは A の残高を減らして，B の残高を増やすという操作を行う．この際，A の残高は減っているが，B の残高は変わっていないという中途半端な操作が起きてはならない．確実に操作全体が成功するか，失敗しなければいけない．このようなトランザクション処理を行うために，高度な機能を提供するのが SOAP，WS-* である．SOAP とは，HTTP などを利用してその他のコンピュータにあるアプリケーションを呼び出すための RPC スタイルのプロトコルである．また，WS-* は SOAP を土台にして様々な仕様を定義したものである．これらの技術は HTTP にはない機能を提供する．例えば WS-AtomicTransaction[46] は 2 相コミットによるトランザクション処理をサポートしている．2 相コミットとは処理に矛盾がないよう整合性を保つ手法である．これによって SOAP Web アプリケーションは容易にトランザクション処理を実現することができる．

以上のように，確かにこのような複雑な動作を行う Web アプリケーションは、高機能な SOAP，WS-*に適している．しかし，トランザクション処理は REST にも可能である．REST ではトランザクション自体をリソースとして作成する．そして，このトランザクションリソースが成功した場合はコミットされ，失敗した場合は削除される．これにより元のリソースの整合性を保つ．また，REST によるトランザクションの実装は利点もある．トランザクションをリソース化することでトランザクション処理の部分がアドレス可能となる．そのため，処理後も参照が可能となり，アーカイブ化やどのような処理を行ったかの確認が可能である．その他にも SOAP に適した Web アプリケーションは存在するが，それらの多くはこのように REST に置換が可能である．

SOAP は REST と相反する Web アプリケーションを作成する．それらは，すべてのリソースが単一の URI で管理され，アドレス可能性のないアプリケーションとなる．また，それぞれのアプリケーションに互換性がないため，REST のように他のアプリケーションをマッシュアップすることができない．これは RPC スタイルをとり，インタフェースに統一性がないことが原因である．このように，SOAP によるアプリケーションは Web の利益をほとんど享受することのできない．それらは不透明で汎用性，拡張性の低いアプリケーションとなる．また，SOAP に適したアプリケーションの多くは REST で補うことが可能である．そのため，Web アプリケーションは利点の多い REST を先に目指すべき

である．どうしても RESTful で実現できない場合のみ REST 以外の実装を考えればよい．これらの理由から本研究も可能な限り WebSocket アプリケーションに REST を適用する．

4.5.2 統合モデルの利点

本研究の提案する統合モデルの最大の利点は，URI による参照が可能となる点である．これは接続性，ステートレス性を向上させて Web アプリケーションの特徴を取り戻すものである．例えば，他ユーザとの電子メールなどでのリソースの共有，ブックマーク，履歴の表示などである．これはユーザにとって最も重要な部分である．また，WebSocket の課題である状態の復帰を解決し，クライアントの再接続の問題にも利用可能である．本統合モデルにより WebSocket アプリケーションはより Web に適したものとなり，ユーザビリティの向上だけでなく，様々な WebSocket アプリケーションの課題を解決する．

第5章　統合モデルの検証

5.1　概要

第4章で提案したRESTとWebSocketの統合モデルを実際にアプリケーションを実装して検証する．本研究ではオセロアプリケーションの実装を行った．統合モデルを適用したオセロアプリケーションとWebSocket通信を実装したのみのオセロアプリケーションとを比較して，本統合モデルの優位性を示す．また，オセロアプリケーション以外のアプリケーションも本統合モデルの適応方法を考察し，評価する．本研究で扱うアプリケーションは単純なものを想定しているが，現実の複雑なサービスもリソースの表現やユーザの動作がリッチになっただけで，設計方法の考え方は変わらない．RESTは複雑なアプリケーションを簡易的にするためのアーキテクチャスタイルである．

5.2　オセロアプリケーションの実装

5.2.1　概要

第4章で提案した統合モデルを適用したオセロアプリケーションの実装を行った．オセロアプリケーションはWebSocketによる非同期通信を行い，オセロゲームを行う．オセロゲームは，8×8マスのボードを使ったボードゲームである．ユーザは白と黒の値を持った2人のプレイヤーと0人以上の観戦者に分かれる．ゲームはプレイヤーが黒と白交互に石を置いていき進んでいく．黒白共にそれ以上打つ所がなくなったらゲーム終了である．サーバサイドはRubyを使用し，WebSocketを実現するためにEM-WebSocketを使用している．Rubyで並列処理を行うためのEventMachineを利用したライブラリで，ハンドシェイクやWebSocket通信のためのメソッドを定義している．クライアントサイドはJavaScriptを使用している．WebSocket API [47] に準拠したものとなっており，WebSocketのために`"onopen"`，`"onmessage"`，`"onerror"`，`"onclose"`の4つのイベントハンドラを持っている．

5.2.2　統合モデル適用前の実装

まず REST を考慮していないオセロアプリケーションを実装を示す．図 5.1 は初期画面である．ユーザが黒，白のプレイヤーか観戦者かを選択するする画面である．例えばユーザが黒を選択すると図 5.2 に画面が遷移し，ゲームが始まる．プレイヤー選択後も同一のリソースと見なされ，URI も変化しない．

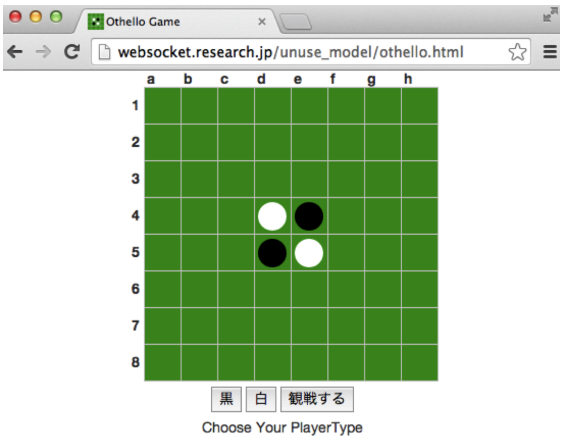


図 5.1: オセロゲームの開始画面

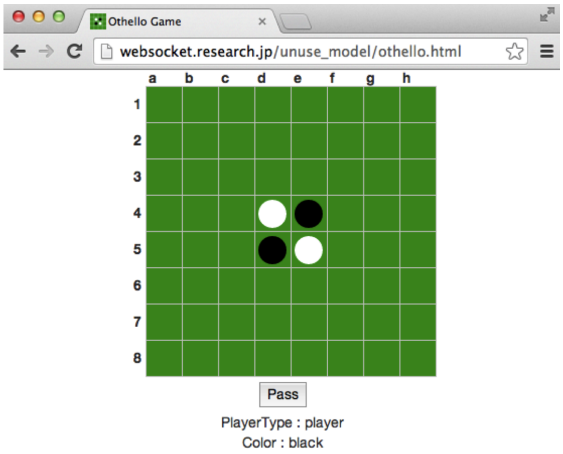


図 5.2: プレイヤーの選択後

ゲームを進めていくと図 5.3 のようになる．図 5.3 はゲーム開始から 5 手進んでいるが，やはり URI は変化しない．そのため，この状態で戻るボタンを押すと 4 手目ではなく，オ

セロアプリケーションに接続した直前の別のページへ飛んでしまう．また，図 5.3 の状態をブックマークすることも，URI で共有することもできない．この URI に再度接続すると図 5.1 の開始画面に飛ぶ．これが REST を考慮していないアプリケーションの例である．このように，REST でないアプリケーションはブラウザの多くの機能を無効にし，URI で共有することができるという Web の利点を享受することができない．そこで，このオセロアプリケーションに本統合モデルを適用する．

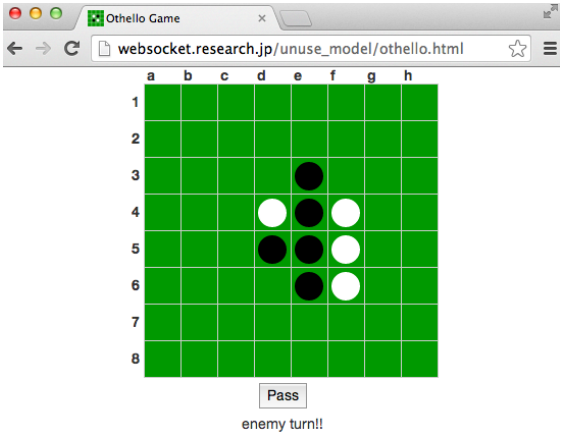


図 5.3: ゲーム途中 (5 手目)

5.2.3　アドレス可能性の追加

オセロアプリケーションのリソースをアドレス可能にする．リソース状態は一手ごとに管理し，それぞれのリソースに Game ID，Player ID，Move を与える．Game ID はゲームの区別，Player ID はプレイヤーの区別，Move は手数の区別に使用する．それぞれの値はクエリとしてデータベースの問い合わせに利用する．例えば Game ID が 54，Player Id が 88，手数が 0 手目の場合，クエリは `"?game_id=54&player_id=88&move=0"` となる．

URI の更新

URI は一手ごとに更新する．しかし，更新のたびにページをリロードするのはユーザの負担が大きく，非同期に通信を行う WebSocket の利点がなくなってしまう．そこで，JavaScript の History API の `pushState` メソッドによって URI を動的に更新する．使用方法は図 5.4 のようになる．今回は履歴にゲームボードの状態を格納し，対応する Game

ID，Player ID，Move の値を持った URI を関連づける．この pushState を一手ごとに呼び出すことで，動的に URI が更新される．

```
history.pushState(state, "", uri);
-----------------------------------
state : イベントオブジェクトに入れるリソース状態
"" : 空文字，null
uri : オブジェクトに対する URI
```

図 5.4: pushState メソッドの利用方法

履歴の呼び出し

pushState で URI とリソースを関連づけ，ブラウザの履歴に格納したので，それを onpopstate メソッドで呼び出す．onpopstate には格納したオブジェクトが引数として与えられる．図 5.5 は実装例である．格納したリソース状態を evt に渡し，evt.state で呼び出している．drawGameBoard はボードの再描画を行う．drawGameBoard の引数は左からボードの状態，手数，パスの回数である．これによって，戻るボタン，進むボタンの使用が可能となる．

```
window.onpopstate = function(evt){
   if(evt.state){
      drawGameBoard(evt.state.board, evt.state.move, evt.state.passed);
   }
};
```

図 5.5: onpopstate メソッドによる履歴の呼び出し

5.2.4 ステートレス性の追加

リソースが URI で一意に識別可能になったため，URI によるリソースの取得を実装する．これで，クライアントは状態の復帰が可能である．HTTP ではリソースの取得を GET で行うが，WebSocket には通信用に send メソッドしか容易されていないため，RPC スタ

イルのメソッドの呼び出しが必要となる．本オセロアプリケーションでは query が存在する場合，データベースからゲームの情報を取得し，それをクライアントに送信するようにした．例えば "?game_id=56&move=5" というクエリを与えた場合，Game ID が 56 の 5 手目が表示される．これでブックマークや URI によるリソースの参照が可能である．

5.2.5 接続性の追加

接続性の追加として，前後のリソースをリンクするようにした．図 5.6 の矢印部分がハイパーリンクとなっている．左矢印が一手前のリソースと接続し，右矢印が一手次のリソースと接続することで，ゲームの棋譜を前後することができる．アドレス可能なので，戻る・進むボタンの実装も URI でリソースを取得するのと同じ動作で実装可能である．

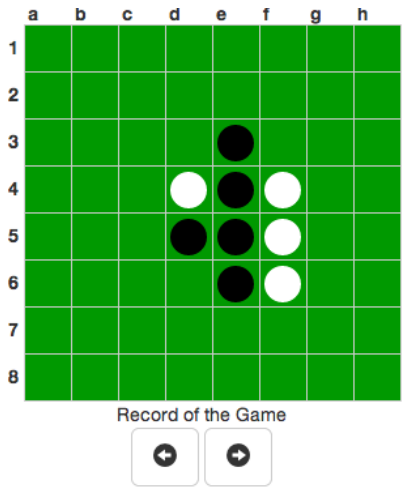


図 5.6: 進む，戻るボタンの実装

5.2.6 アプリケーションの動作

オセロアプリケーションにアドレス可能性を付加し，それによってステートレス性，接続性を向上させた．統合モデルを適用後のアプリケーションは次の動作を取る．

まず，ユーザがプレイヤーを選択すると図 5.7 のように "?game_id=57&player_id=92&move=0" というクエリが与えられている．Game ID はゲームの情報，Player ID はプレイヤーの情報を管理している．ゲーム開始前なので Move は 0 である．統合モデル適用前の図 5.2 では URI は変わらなかった．

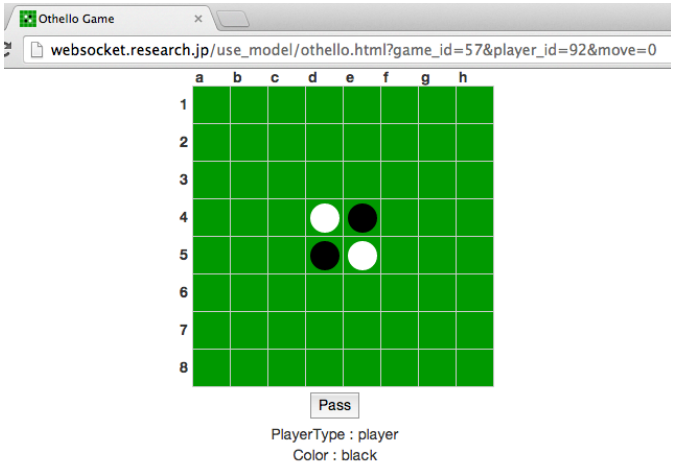


図 5.7: 統合モデルを適用した際のプレイヤー選択後の画面

次に図 5.8 はゲームを進めていった図である．図 5.7 と比較すると Move の値が変わっているのがわかる．現在 5 手目まで進んだので"move=5"となっている．統合モデル適用前の図 5.3 では URI は変化していなかった．この URI はブックマークや参照が可能である．そして，戻るボタンを押すと 4 手目に戻ることができる．ここで，実際に "http://websocket.research.jp/use_model/othello.html?game_id=57&player_id=92&move=5" にアクセスすると図 5.9 に遷移する．下の矢印ボタンで前後のリソースに遷移することも可能である．統合モデル適用前は URI で飛ぶことも，リソースをハイパーリンクで関連づけることもできなかった．これによって，ユーザは自分の行きたいゲーム，手数に遷移することが可能である．

5.3 オセロアプリケーションの考察

WebSocket で通信するオセロアプリケーションに本統合モデルを適用し，アドレス可能性，ステートレス性，接続性を付加した．その結果，ユーザが一手打つたびに URI が更新され，その URI をブックマークや参照することが可能になった．また，URI が更新されることでブラウザの履歴に格納することが可能となり，戻る・進むボタンが利用できるようになった．これは，ユーザビリティを大きく向上させている．また，ユーザは URI によってリソースを推測可能となっている．一手打つたびに Move の値が増えていくので，Move と手数が結びつくことが容易に推測できる．これによって，ユーザは手元にない URI の

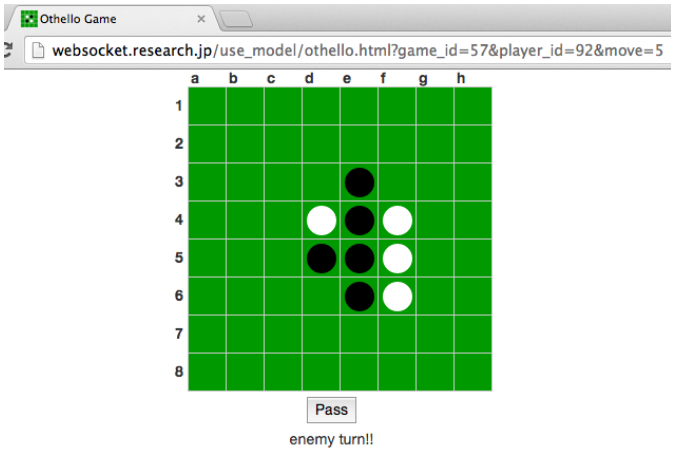


図 5.8: 統合モデルを適用後のゲーム (5 手目)

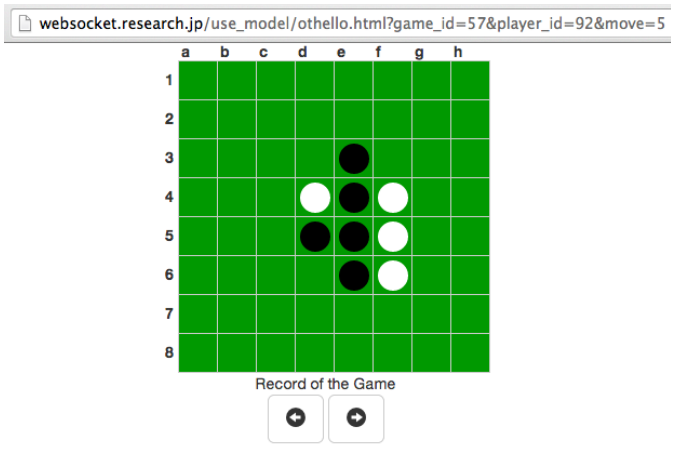


図 5.9: URI による履歴の表示

リソースに飛ぶことも可能である．今回は簡易化のために Game ID，Player ID，Move をすべてクエリに指定しているが，本来 Game ID と Move のみ棋譜の表示は可能である．Player ID のみを指定することでプレイヤーの情報を表示するページなども容易に実装可能である．リソースをアドレス可能にすることで，多くの動作がハイパーリンクのみで実装可能である．これは，開発者の実装コストを下げ，ユーザにはわかりやすい仕組みである．また，セキュリティ面などを気にする必要のないアプリケーションの場合，再接続の際も URI で状態復帰が可能である．動作の差を表にまとめると表 5.1 のようになる．

表 5.1: 統合モデル適用前と後の動作の比較

動作	統合モデル適用前	統合モデル適用後
非同期処理	○	○
URI によるリソースの参照	×	○
URI による状態の復帰	×	○
ブックマーク	×	○
戻るボタンの使用	×	○
ハイパーリンクによる接続	×	○

5.4 その他のアプリケーション

本研究ではオセロアプリケーションの実装を行い，統合モデルが実装可能であることを示した．ここでは，オセロアプリケーション以外のアプリケーションにどのように適用すれば良いかを考察する．これによって，更なる統合モデルの有意性と実現可能性を示す．

5.4.1 その他のボードゲーム

オセロ以外のボードゲームについて考察する．例えば，将棋や囲碁，チェスである．これらはオセロよりもコマの動きが多様で，状態数の多い複雑なゲームである．しかし，一手ごとに離散的にリソースの状態が変化することは変わらない．また，状態数も 1 ゲーム 100，200 程度である．そのため，オセロゲームと同様に統合モデルを適用可能である．また，すごろくのようなゲームも一手ごとにイベントが発生して，状態が変化するという点でオセロと同様である．ゲームのルールは大きく違うが，同様に統合モデルに適用可能である．よって，ボードゲームは本統合モデルに適用可能である．

5.4.2 株価チャート

WebSocket で想定される代表的なアプリケーションの一つとして，株価チャートアプリケーションがある．株は情報のリアルタイム性に対する価値が高く，秒単位でデータが更新される．また，多くのユーザが接続を維持し続ける．そのため，WebSocket に適したアプリケーションとなっている．今回はある単一の株銘柄に対して，横軸に時間，縦軸に価格を取る単純な株価チャートについて考察する．

リソース状態はデータが更新されるたびに管理すべきである．株価の更新は日時で管理が可能なので，`?date=` などというクエリでアクセス可能とすればよい．これによって，すべてのリソースを参照することができ，リソースの行き来が可能となる．ユーザは特定の状態をハイパーリンクでアーカイブ可能である．ここまではオセロとほぼ同様だが，問題は URI の更新である．オセロと違い株価は情報の更新頻度が高い，長時間接続するとリソースの状態数が相当数になってしまう．そこで，URI の更新には何らかの工夫が必要である．例えば，株価が一定の変動率を超えたときやユーザの売買のときなど，何らかのイベントが発生したときである．意図を持たない一定時間による更新は避けるべきである．目的はユーザを任意の状態に戻れるようにすることであり，単純な時間による更新はむやみにリソースを増加させる．

以上が統合モデルの適用である．株価チャートはオセロに比べて状態数が多いが，URI の更新に意味を持たせることで問題なく適用可能である．クエリを工夫することで，さらに複雑な株価チャートを実現できる．

5.4.3 地図ナビゲーション

スマートフォンの普及で GPS を利用したルート案内サービスが頻繁に利用されている．それらの多くはネイティブアプリケーションであるが，今後は WebSocket を使用することで Web でのリアルタイムナビゲーションも可能である．想定する機能はマップの表示，現在の居場所の表示，目的地までのルートの表示である．ユーザの動きに合わせて，リアルタイムにルートが更新される．

まずオセロアプリケーションと同様に適用させる．リソースはデータとして，地図，ユーザの座標，目的地を持つ．そこで，クエリとしてユーザの座標と目的地の座標を持たせれば状態の復帰が可能である．その場合 URI の更新はユーザの座標が変わったときとなる．これで十分統合モデルは適用可能である．しかし，問題が二つある．更新頻度の高さと履歴の保存の有意性である．普通ユーザは道案内サービスを利用する場合，現在のルート情

報が大切であり，過去の状態に戻ることはまれである．そのため，ユーザの位置に合わせ URI を更新させることは無駄である．そこで，特殊なイベントによって URI を更新させる．これにより，他ユーザとのルートの共有など際に利用可能である．現在のデータの利用価値が高く，過去のデータの利用価値が低い場合は自動に更新させずに，ユーザが必要としたときにのみ更新させる．

5.4.4 まとめ

REST と WebSocket アプリケーションとの統合モデルは多くの場合に適用可能である．特に離散的でリソース状態の変化が少ないアプリケーションは適用しやすい．また，状態数がアプリケーションも十分適用可能だが，その際はリソースの扱いと URI の更新に注意が必要である．リソースはユーザが参照に値するもののみリソースとして扱い，URI を更新させる．そうすることで，適切にユーザがリソースの状態を遷移できるようになる．

第6章　おわりに

6.1　まとめ

本研究の目的は，WebSocket アプリケーションの REST への対応とそれに伴う WebSocket の課題の解決である．そのために，本研究では REST と WebSocket アプリケーションの統合モデルの提案とその検証を行った．

WebSocket は Web でステートフル通信を行う通信技術である．これはリアルタイム性に優れ，小さいオーバヘッドでリアルタイム Web を実現することができる．しかし，ステートフル性は Web で現在使われている多くの機能を無効にする．例えば，ブックマークや履歴による戻る機能，URI による参照である．これらの機能はユーザビリティにとても重要である．そこで，本研究では REST との統合モデルを提案し，これらの機能を WebSocket アプリケーションで使用可能な設計方法を示した．また，その実現性や有効性を具体的なアプリケーションを挙げて検証を行った．本統合モデルは多くのアプリケーションに適用可能であり，より Web に適したリソース指向のアプリケーションの実装を可能とする．そして，上記の課題を解決し，WebSocket アプリケーションの課題である状態の復帰を解決する．

6.2　今後の課題

本研究では REST の基準として ROA を採用している．ROA は RESTful の条件にアドレス可能性，ステートレス性，接続性，統一インタフェースを定義している．ただし本研究の統合モデルはアドレス可能性，ステートレス性，接続性に対してのみ言及している．統一インタフェースは多くの実装例を元に提案し，多くの立場の人達との議論が必要な要素である．そのため，本研究では取り扱わなかった．しかし，統一インタフェースは Web の多様性，汎用性，拡張性を向上する重要な制約である．RESTful Web サービスは外部の API をマッシュアップすることが可能である．これは，統一インタフェースによって常に HTTP メソッドが同じ動作をするためである．

RPC スタイルのアプリケーションの場合，各自でメソッドを定義しているため，使用可能なメソッドをアプリケーションごとに把握しなければならない．これは，ユーザにとって大きな負担である．現在，WebSocekt が通信に使用するメソッドは `send` メソッドのみである．そのため，HTTP の `GET`，`POST`，`PULL`，`DELETE` のようなメソッドが必要である．今後の課題は，より多くのアプリケーションを検証し，それによって得られる共通部分から統一インタフェースの提案を行うことである．

謝辞

本論文を執筆するにあたり，多くの方々からご御指導及び御助言を賜りました．素晴らしい研究テーマと環境を与えてくださり，丁寧に御指導して頂いた Martin J. Dürst 教授に深くお礼申し上げます．この論文を何度も校正していただいた松原俊一助教に心から感謝いたします．ご多忙のところ副査を担当していただいた大原剛三教授，戸辺義人教授にお礼申し上げます. 数々の話に付き合っていただき，また意見を交わし，お互いに刺激しあってきた研究室の方々に感謝いたします．WebSocket プロトコルの策定者である Ian Fette をはじめとする開発者の皆様に感謝いたします．毎日のように長時間作業していても，いつも笑顔で対応して頂いたカフェの店員の方々ありがとうございます．最後に，本研究の遂行にあたり助力をいただいたすべての方々に，心から感謝申し上げます.

2014 年 1 月

石畑 翔平

参考文献

[1] Ian Fette and Alexey Melnikov. The WebSocket Protocol. RFC 6455 (Proposed Standard), December 2011.

[2] Jon Postel. Transmission Control Protocol. RFC 793 (Internet Standard), September 1981.

[3] Theodore John Socolofsky and Claudia Jeanne Kale. TCP/IP tutorial. RFC 1180 (Informational), January 1991.

[4] Jon Postel. User Datagram Protocol. RFC 768 (Internet Standard), August 1980.

[5] Brian Carpenter, Stuart Cheshire, and Robert M. Hinden. Representing IPv6 Zone Identifiers in Address Literals and Uniform Resource Identifiers. RFC 6874 (Proposed Standard), February 2013.

[6] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986 (Internet Standard), January 2005.

[7] Martin Dürst and Michel Suignard. Internationalized Resource Identifiers (IRIs). RFC 3987 (Proposed Standard), January 2005.

[8] Uniform Resource Identifier (URI) Schemes. `http://www.iana.org/assignments/uri-schemes/uri-schemes.xhtml`.

[9] Paul V. Mockapetris. Domain names - concepts and facilities. RFC 1034 (Internet Standard), November 1987.

[10] Paul V. Mockapetris. Domain names - implementation and specification. RFC 1035 (Internet Standard), November 1987.

[11] Roy T. Fielding, James Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999.

[12] Mark Nottingham and Roy T. Fielding. Additional HTTP Status Codes. RFC 6585 (Proposed Standard), April 2012.

[13] Adam Barth. HTTP State Management Mechanism. RFC 6265 (Proposed Standard), April 2011.

[14] Alex Russell. Comet: Low Latency Data for the Browser. `http://infrequently.org/2006/03/comet-low-latency-data-for-the-browser/`, March 2006.

[15] Tim Berners-Lee, Roy T. Fielding, and Henrik Frystyk Nielsen. Hypertext Transfer Protocol – HTTP/1.0. RFC 1945 (Informational), May 1996.

[16] Roy T. Fielding, James Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, and Tim Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2068 (Proposed Standard), January 1997. Obsoleted by RFC 2616.

[17] Httpbis Status Pages. `http://tools.ietf.org/wg/httpbis/`.

[18] Roy T. Fielding and Richard N. Taylor. Principled design of the modern Web architecture. *ACM Transactions on Internet Technology (TOIT)*, Vol. 2 Issue 2, pp. 115–150, May 2002.

[19] Marc Hadley, Yves Lafon, Jean-Jacques Moreau, Henrik Frystyk Nielsen, Martin Gudgin, Anish Karmarkar, and Noah Mendelsohn. SOAP version 1.2 part 1: Messaging framework (second edition). W3C Recommendation, W3C, April 2007. http://www.w3.org/TR/2007/REC-soap12-part1-20070427/.

[20] Google Apps Platform — Google Developers. `https://developers.google.com/google-apps/documents-list/`.

[21] Documentation - Twitter Developers. `https://dev.twitter.com/docs`.

[22] Mark J. Handley, Eric Rescorla, and IAB. Internet Denial-of-Service Considerations. RFC 4732 (Informational), December 2006.

[23] Leonard Richardson and Sam Ruby. *Restful Web Services*. Oreilly & Associates Inc, 2007.

[24] Tim Berners-Lee. Cool uris don't change. `http://www.w3.org/Provider/Style/URI.html`, 1998.

[25] Robert Thurlow. RPC: Remote Procedure Call Protocol Specification Version 2. RFC 5531 (Draft Standard), May 2009.

[26] StatCounter Global Stats. `http://gs.statcounter.com/`.

[27] Market share for mobile, browsers, operating systems and search engines - NetMarketShare. `http://netmarketshare.com/`.

[28] Paul J. Leach, Michael Mealling, and Rich Salz. A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122 (Proposed Standard), July 2005.

[29] Acting Director Patrick Gallagher. Secure Hash Standard. FIPS PUB 180-3, October 2008.

[30] Simon Josefsson. The Base16, Base32, and Base64 Data Encodings. RFC 4648 (Proposed Standard), October 2006.

[31] Donald E. Eastlake 3rd, Jeffrey I. Schiller, and Steve Crocker. Randomness Requirements for Security. RFC 4086 (Best Current Practice), June 2005.

[32] Bradford G. Nickerson Victoria Pimentel. Communicating and Displaying Real-Time Data with WebSocket. *IEEE Internet Computing Volume 16 Issue 4*, July 2012.

[33] Takeshi Yoshino. Compression Extensions for WebSocket, draft-ietf-hybi-permessage-compression-17, January 2014.

[34] L. Peter Deutsch. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951 (Informational), May 1996.

[35] Juliano Rizzo. The CRIME attack. `https://docs.google.com/presentation/d/11eBmGiHbYcHR9gL5nDyZChu_-lCa2GizeuOfaLU2HOU/edit#slide=id.g1eb6c1b5_3_6`.

[36] Takeshi Yoshino. A Multiplexing Extension for WebSockets, draft-ietf-hybi-websocket-multiplexing-11, July 2013.

[37] The WebSocket Application Messaging Protocol. `http://wamp.ws/`.

[38] Mark Birbeck and Shane McCarron. CURIE Syntax 1.0. W3C Working Group Note, December 2010. http://www.w3.org/TR/2010/NOTE-curie-20101216.

[39] Realtime Framework. `http://www.realtime.co/developers/realtimeframework`.

[40] xRTML.org: The language for the real time web. `http://www.xrtml.org/`.

[41] xRTML API. `http://docs.xrtml.org/xrtml/javascript/3-2-0/xrtml.html`.

[42] EM-WebSocket. `https://github.com/igrigorik/em-websocket`.

[43] Socket.IO. `https://github.com/igrigorik/em-websocket`.

[44] Autobahn. `http://autobahn.ws/`.

[45] Anish Karmarkar, Jean-Jacques Moreau, Marc Hadley, Noah Mendelsohn, Martin Gudgin, Henrik Frystyk Nielsen, and Yves Lafon. SOAP version 1.2 part 1: Messaging framework (second edition). W3C Recommendation, W3C, April 2007. http://www.w3.org/TR/2007/REC-soap12-part1-20070427/.

[46] Luis Felipe Cabrera (Microsoft), George Copeland (Microsoft), Max Feingold (Microsoft), Robert W Freund (Hitachi), Tom Freund (IBM), Jim Johnson (Microsoft), Sean Joyce (IONA), Chris Kaler (Microsoft), Johannes Klein (Microsoft), David Langworthy (Microsoft), Mark Little (Arjuna Technologies), Frank Leymann (IBM), Eric Newcomer (IONA), David Orchard (BEA Systems), Ian Robinson (IBM), Tony Storey (IBM), and Satish Thatte (Microsoft). Web Services Atomic Transaction (WS-AtomicTransaction) Version 1.1, July 2007.

[47] Ian Hickson. The WebSocket API. Candidate Recommendation, W3C, September 2012. http://www.w3.org/TR/2012/CR-websockets-20120920/.

付録

A. 質疑応答

付録 A
質疑応答

発表後の質疑応答

Lopez 先生の質問

リソースの定義で重要なものよりも，重要でないものに注目する方が大切だと思うのですがどうですか？

回答

例えばチャットを発言がないのにもかかわらず，一秒ごとにリソースとしていたら無駄です．このように，重要でないものは無尽蔵にあります．そのため，重要でない情報に注目することには意味がなく，後にそのアプリケーションが使う情報，つまり参照する情報をリソースと定義する方がシンプルでわかりやすくなります．

戸辺先生の質問

そもそも REST に対応する必要があるのか，実際には Web にも Cookie があったりと，完全にステートレスでない状態なのに，本当に REST という必要があるのか？

回答

確かに WebSocket はステートフルであり，REST に適さない通信方法です．そのため，WebSocket は REST に対応させるべきではないという意見もあります．しかし，私は WebSocket が Web アプリケーションのための通信技術である以上，REST に対応するべきだと考えます．その理由は発表で挙げた利点にあります．汎用的な WebSocket アプリケーションを開発するためには REST への対応が必要です．また，ステートレスの例に Cookie を挙げていますが，Cookie の使用は RESTful アプリケーションでは認められています．ただし，クライアントがステートレス性を失わないように，その使用方法には制限があります．

小宮山先生の質問

Flash などは一つの URI でサービスを利用することができるが，REST にすることでページ数などの情報が増えてしまうことは問題ではないのか？

回答

確かにそのような欠点はありますが，REST にすることで得られる利点の方が大きいので，多くの場合で REST が採用されています．ただし，REST 以外のアーキテクチャの方が簡単に実装することができる場合は，その他のアーキテクチャに沿って実装されます．例えばインターネットバンクなどのサービスがあります．これらは各会社だけで使われるガラパゴス的なサービスなので，汎用性が大きな利点とならないためです．このような一部の Web アプリケーション以外は REST に沿って開発されています．

フォーラムの質疑応答

齊藤先生の質問

地図アプリケーションについて質問です．リソースの保存の仕方を三つ検討していましたが，それぞれの評価 (リソースの量，アンケートなど) はできませんか．

回答

もちろんそれぞれの評価は可能です．実際に多くの企業は AB テストなどを使い，そのような評価を行っています．ただし，それぞれのサービスによって最適な方法は変わってくるので，統合モデルにおいて個々のアプリケーションの実装方法の評価はそれほど大きな価値を持ちません．

松村先生の質問

Web のエンドユーザに対してどういう貢献があるか分かりやすい発表になっていたと思います．ちょっと聞き逃したのかもしれませんが，統合モデルを提案したうえで，何をリソースとするか，更新タイミングをどう設定するか，といったときに考えるべき指針を示したということでよいでしょうか？

回答

ありがとうございます．提案した統合モデルをアプリケーションに適用させる際に，リソースの扱いや更新基準が必要ということです．リソースの扱いは，リソース指向アプリケーションの核となる部分です．

Lopez 先生の質問

今回提案している手法では各ウェブアプリケーションに合わせて URI に用いるタグを変えなければいけないが，もっと汎用的な方法が検討できないでしょうか？例えば日付タグとか．

回答

確かに多くのリソースは日付けによる管理が有効な可能性があります．そのため，様々なアプリケーションによる検証が必要ですが，ライブラリの開発の際にはそのような具体的な方法を示すのは適しているかもしれません．しかし，今回の統合モデルは具体的なアプリケーションに限定するもではないので，URI の更新を日付けなどに制限することは，かえって汎用性を失わせてしまいます．

盛川先生の質問

提案されたモデルはどこまで一般化されたものなのでしょうか．例えば，アプリケーションごとに実装方法を検討する必要があるのだとすると，ここでは例として挙げたオセロアプリにしか適用できない手法であるとは言えませんか．もしくは実装の指針を示したということでしょうか．得られた知見の適用範囲を示してもらいたいです．

回答

適用範囲は WebSocket アプリケーション全体です．本研究では実際に，その他のボードゲーム，地図アプリケーション，株価アプリケーションについて検証しました．しかし，ご指摘の通り適用できないアプリケーションも考えることができるので，どのようなアプリケーションが適さないのかをさらに検証していく必要があります．