卒業論文

サーバプッシュを有効に活用した Webアクセス高速化方式

2020年1月24日

八木橋 晃一

学籍番号　CY16173

指導教員　山崎 憲一

芝浦工業大学

デザイン工学部　デザイン工学科

概 要

近年インターネットは身近なものとなり，個人がWebページにアクセスする機会が増えている．インターネットの普及に伴い，Webページを構成するコンテンツも他種多様となり，その種類や機能も多岐に渡る様になった．しかし，コンテンツの数や大きさの増加は，ユーザがWebページを表示させる時間までの増加につながる．現在では，コンテンツが溢れており他のコンテンツで代用可能な場合が多い．そのため，画面表示に時間を要してしまうとユーザは別のコンテンツへと移動してしまう．Webページを閲覧する際に,画面表示にかかる時間は閲覧者数を左右する大きな要因となる．Webアクセスにおいては，クライアント（ブラウザ）がHTMLのリクエストをサーバに送信し，サーバはレスポンスとしてHTMLを返す．クライアントはHTMLを解析し，画面表示に必要となるCSSなどのリクエストをサーバに送信し，サーバからそれらを受け取ると画面を表示する．つまり，クライアントはHTMLを手に入れてから再度サーバに要求を出し，そのレスポンスが返ってくるまで描画を始めることができない．近年提案されたWeb通信プロトコルHTTP/2にはサーバプッシュと呼ばれる機能がある．プッシュを利用するためには何を，いつ，どの様な順番でプッシュすべきかを検討する必要がある．本研究ではブラウザが設定したコンテンツの優先度を用いてプッシュを制御する手法を提案する．サーバ側で要求を記録することで，そのWebページを構成するのに必要なファイルを把握するとともに，ブラウザ側の優先度を理解することで描画に必要のないファイルを適切に判断する．これにより，画面の描画に必要なファイルを優先してプッシュすることができる．この様な提案システムを実装するにあたって，技術課題を解決し，その詳細な実装について述べる．提案したシステムを実装したサーバで，複数のコンテンツを用いてFirst PaintとPage Load Timeを測定して評価を行い，複数の問題点はあるものの往復通信回数の削減に有用であることを示す.

目 次

表 目 次

図 目 次

1 はじめに

1.1 研究背景

Webアクセスにおいては，クライアント（ブラウザ）がHTMLのリクエストをサーバに送信し，サーバはレスポンスとしてHTMLを返す．クライアントはHTMLを解析し，画面表示に必要となるCSSなどのリクエストをサーバに送信し，サーバからそれらを受け取ると画面を表示する．

近年提案されたWeb通信プロトコルHTTP/2にはサーバプッシュと呼ばれる機能がある．これはクライアントからのリクエストなしに，サーバがデータを送信する機能である．クライアントがHTML解析後にCSSなどをリクエストすることは，HTML所有者であるサーバは事前に把握可能である．最初のHTMLのリクエストが来た段階で，それらをプッシュすれば，クライアントはHTMLの解析終了後ただちに画面を表示できる．この2つのデータフローの概略を図1.1に示す．

Webページを閲覧する際に,画面表示にかかる時間は閲覧者数を左右する大きな要因となる. 文献[1]では0.1秒の遅延が発生すると売上が約1%減少したという報告がある.また，文献[2]では画面の表示が1秒遅れることで，年間16億ドルの損失が生じるという報告がある．以上のことから，画面表示にかかる時間を短くすることは，僅かであっても大きな効果が期待できる．

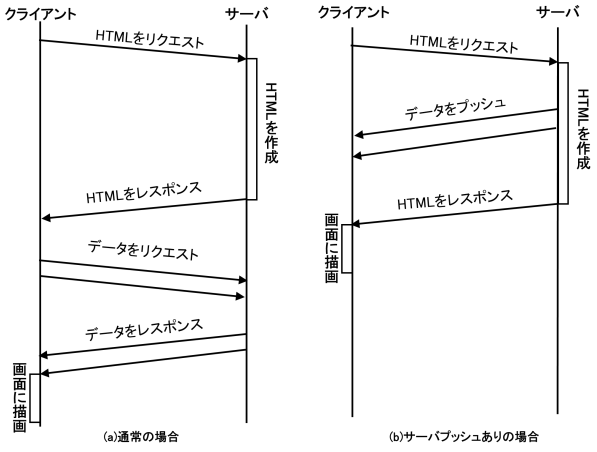

図 1.1 通信フローの概略図

1.2 研究目的

サーバプッシュはメカニズムとして定義されているが，何をいつプッシュすべきかは十分検討されていない[3]．本研究ではWebパフォーマンスを向上させるためのサーバプッシュ方式を提案する．なお本研究でのWebパフォーマンスとは，ブラウザがWebページを表示する時間の短さを指す．

文献[4]では，CSS, JavaScript, 画像の優先順位で，サーバ側がHTMLを作成している間だけプッシュすることを提案している．しかし，この優先度が全てのWebページにおいて適切とは限らず，描画には必要のないファイルをプッシュしている可能性がある．

また，HTMLの作成が終わるまでに，描画に必要なファイルを全てプッシュできないという場合があるという問題がある．その場合，クライアントがHTML解析後にそのファイルの要求を出し，サーバが返すまで描画を開始することができない．

文献[5]ではそのWebページを構成するファイルを全てプッシュすることを提案している．これにより，クライアントがHTMLファイルを受け取った段階で，描画に必要なファイルが手元にある状態にすることができる．しかし，最後にHTMLを受け取るまで，クライアントは描画を始めることができないという問題がある．

本研究では描画に必要のないファイルを判断することで，描画に必要なファイルを全てプッシュする手法を提案する．

1.3 論文構成

本論文は次のような構成となっている．まず，第2章では関連研究と技術課題について述べ，第3章で本研究の提案手法を述べる．第4章では，開発環境と提案システムの実装について述べる．第5章では本研究の提案手法，先行研究，サーバプッシュなしのサーバで比較実験を行い，結果の考察を行う．最後に第6章では本研究のまとめを述べる．

2 関連研究

本章ではまず，本研究で利用した関連技術について述べる．次に先行研究を2つ取り上げ，それぞれの改善点について述べる．

2.1 関連技術

2.1.1 サーバプッシュ

サーバプッシュのメカニズムは文献[6]に示されている．サーバプッシュのデータフローを図2.1，2.2に示す．サーバプッシュはリクエストに対してサーバが行う．サーバプッシュ行うためにはリクエストに対してPUSH_PROMISEフレームを送信する必要がある．PUSH_PROMISEフレームはクライアントが開始したストリーム上でサーバから送信される．そのため，そのストリームはサーバにおいて“open”または“half-closed （remote）”のいずれかの状態でなければならない．つまり，リクエストを返してから，そのストリーム利用してPUSH_PROMISEフレームを送信することはできない．PUSH_PROMISEフレームは，リクエストヘッダーフィールドの完全なセットからなるヘッダーブロックとプッシュを予約するストリームIDを含む．予約するストリームIDはサーバで使用可能なストリームIDから選択される．

PUSH_PROMISEフレームを受けとったクライアントはプッシュを許可するか，許可しないかを選択することができる．許可しない場合はPROTOCOL_ERRORとして応答する．許可した場合，クライアントは予約済みストリームが終了するまで，そのレスポンスに対してリクエストを発行することは推奨されていない．ただし，サーバが予約済みレスポンスの送信開始まで時間がかかりすぎる場合などには，クライアントはRST_STREAMフレームを送信することでプッシュの予約を取り消すことができる．これには予約済みのストリームIDを参照するため，CANCELまたはREFUSED_STREAM コードのいずれかを使用する．PUSH_PROMISEフレームの送信により，プッシュするための新しいストリームが生成される．PUSH_PROMISEフレームを送信後，サーバは予約されたIDのストリーム上でレスポンスと同様にプッシュレスポンスの送信を始める．プッシュされたデータはキャッシュに保存される．

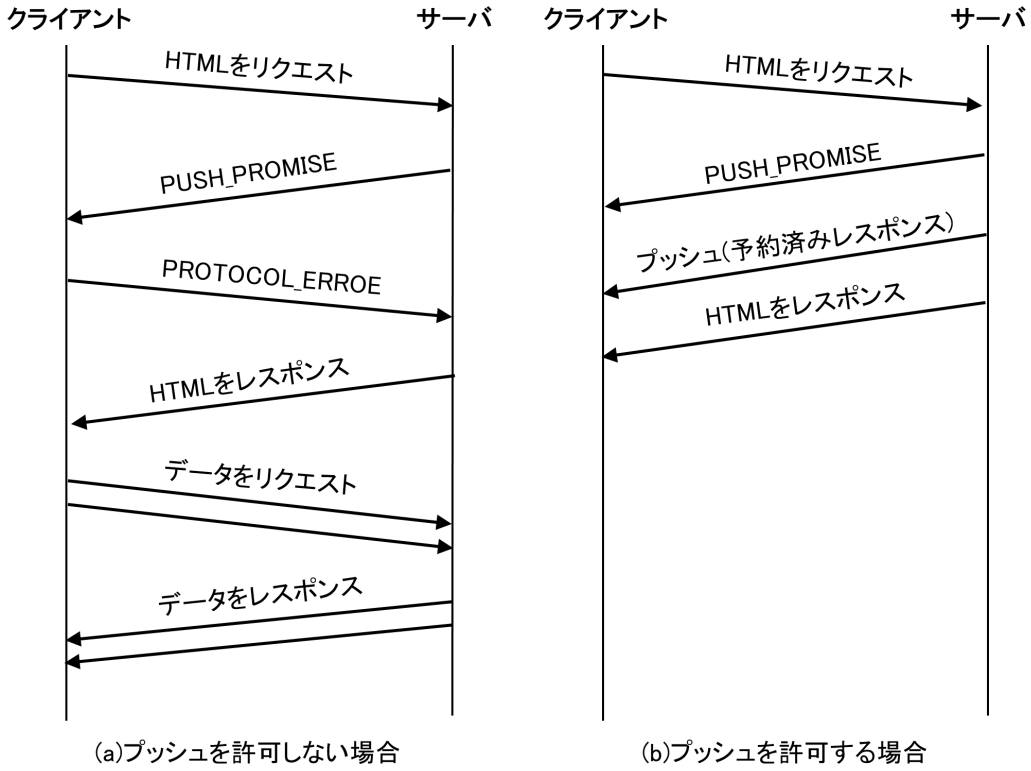

図 2.1 PUSH_PROMISEを許可しない場合と許可する場合のデータフロー図

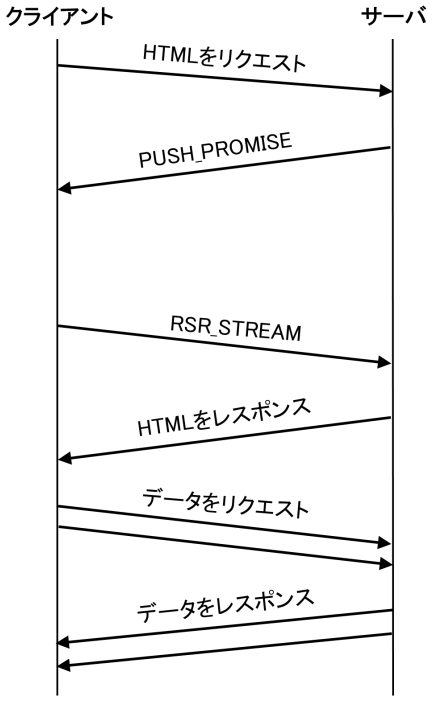

図 2.2 RST_STREAM送信時のデータフロー図

2.1.2 優先度づけ

クライアントは，ストリームに対して優先度をつけることができる．優先度づけのメカニズムは文献[6]に示されている．クライアントはストリームを開始するHEADERSフレームに優先度情報を含めることで，新しいストリームの優先度を指定できる．この優先度はPRIORITYフレームを送信することで，いつでもストリームの優先度を変更することができる．送信キャパシティが限られている場合，フレームを転送するストリームを選択するための指標にすることが可能である．優先度は他のストリームに依存する関係を作ることで設定する．

各依存関係には相対的な重みが割り当てられる．この重みの値はストリームにあるweightに1から256までの整数で設定される．weightは同じストリームに依存するストリームに割り当てるリソースを決定するために使用される．割り当てられるリソースは相対的な比率である．例えば，weightの値が4であるストリームAと，weightの値が12であるストリームBがストリームCに依存しているとする．ストリームCに何も進展がなければ，ストリームAにはストリームBに割り当てられたリソースの3分の1が割り当てられる．優先度はあくまで提案であり，サーバは必ずしも従う必要はない．優先度を設定しない場合，weightの値にはデフォルトの値である16が設定される．

2.1.3 CSSの用途の指定と優先度

CSS（Cascading Style Sheets）とは，Webページのスタイルを設定するコードである．CSSはHTML 文書の要素に選択的にスタイルを適用スタイルシート言語であり，プログラミング言語ではない[7].

また，CSSにはメディアクエリを設定できる．メディアクエリは，プリンタや画面といった端末の種類に合わせて，画面の解像度やブラウザ上での幅などを設定することができる[8]．図2.3にメディアクエリの使用例を示す.

通常CSSの優先度は高いが，ブラウザはメディアクエリの記述を基に，一部のCSSの優先度を他のCSSよりも低く設定する．優先度が低く設定されるCSSの例としては，メディアクエリがprintのものや，クライアントのscreenと異なるものが挙げられる．

```
<!-- メディアクエリの例 -->
<link rel="stylesheet" href="example1.css" media="all" />   <!-- 全てのメディア -->
<link rel="stylesheet" href="example2.css" media="print" /> <!-- プリンタ -->
<link rel="stylesheet" href="example3.css" media="only screen and (max-width: 640px)" > <!-- 幅を指定 -->
```

図 2.3 メディアクエリの使用例

2.1.4 JavaScriptの実行タイミング

ブラウザはHTMLを受け取るとHTMLを解析する．HTMLの解析が完了直後にDOMContentLoadedと呼ばれるイベントが発火する[9]．また，HTMLの解析が完了し，CSSや画像などのダウンロードと表示，JavaScript ファイルのダウンロードと実行など，全てのリソースの処理が完了した直後にloadと呼ばれるイベントが発火する[10]．時系列では描画を始めるのはDOMContentLoadedイベントより後であり，描画が完了するのはloadイベントより後である．

JavaScriptとは動的なコンテンツの更新，マルチメディアの管理，その他多くのことができるスクリプト言語である[11]．HTMLからJavaScriptのファイルを参照する場合，参照先に<script>タグをつける必要がある．図2.4に示すように，HTML5では<script>タグに対して，asyncやdeferといった属性を付与することができる．属性を付与することにより，JavaScriptのダウンロードと実行のタイミングを設定することができる．タイミングの詳細については文献[12]に示されている．それぞれのタイミングについて図2.5に示す．ただし，それぞれのJavaScriptのダウンロードと実行のタイミングの比較がしやすいように，HTMLの解析に要する時間を統一せずに示している．JavaScriptのダウンロードと実行のタイミングは本研究に大きく関係するため，以降では詳細に説明する．

<script>タグにasync，deferのどちらも付けない場合，同期的に読み込む．すなわち，HTMLの解析中に<script>タグにたどり着き次第，解析を一時中断し，JavaScriptファイルのダウンロードと実行を行う．JavaScriptファイルの実行が完了するまでHTMLの解析は再開されない．

<script>タグにasync属性を追加した場合，非同期的にJavaScriptファイルをダウンロード，実行する．ダウンロードはHTMLの解析などを行うメインスレッドとは別のスレッドで実行される．そのため，ダウンロードによって，HTMLの解析を止めることなく，並行してダウンロードを行うことができる．ダウンロードが完了

次第JavaScriptファイルが実行される．しかし，JavaScriptファイルの実行自体は同じメインスレッドで行われたため，実行中はHTMLの解析が一時的に停止する．また，複数の<script>タグにasync属性を追加した場合，それらの実行順序は<script>タグの記述順ではなく，ダウンロードが完了したものから順に実行される．図2.5では，HTMLの解析が完了する前に実行を開始しているが，図2.6に示すように，DOMContentLoadedイベント発火後に実行を開始する場合もある．ただし，JavaScriptファイルの実行の完了がloadイベント発火の条件に含まれているため，loadイベントより前に実行される．

<script>タグにdefer属性を追加した場合，HTMLの解析完了後，DOMContentLoadedイベントの直前にJavaScriptファイルが実行される[12]．実行はHTMLの完了後であるが，ダウンロードはasync同様<script>タグが解析され次第，非同期的にダウンロードが行われる．また，asyncとは異なり複数の<script>タグがdefer属性で記述されている場合，<script>タグの記述順に実行される．

JavaScriptに属性を追加した場合と，追加しなかった場合ではweighの値が異なる．属性を追加しなかった場合の方が，大きいweighの値が設定される．asyncとdeferのweightの値は同等である．

```
<!-- JavaScriptの属性設定の例 -->
<script src="example1.js"></script>          <!-- default -->
<script src="example2.js" async></script>    <!-- async -->
<script src="example3.js" defer></script>    <!-- defer -->
```

図 2.4 async/deferの使用例

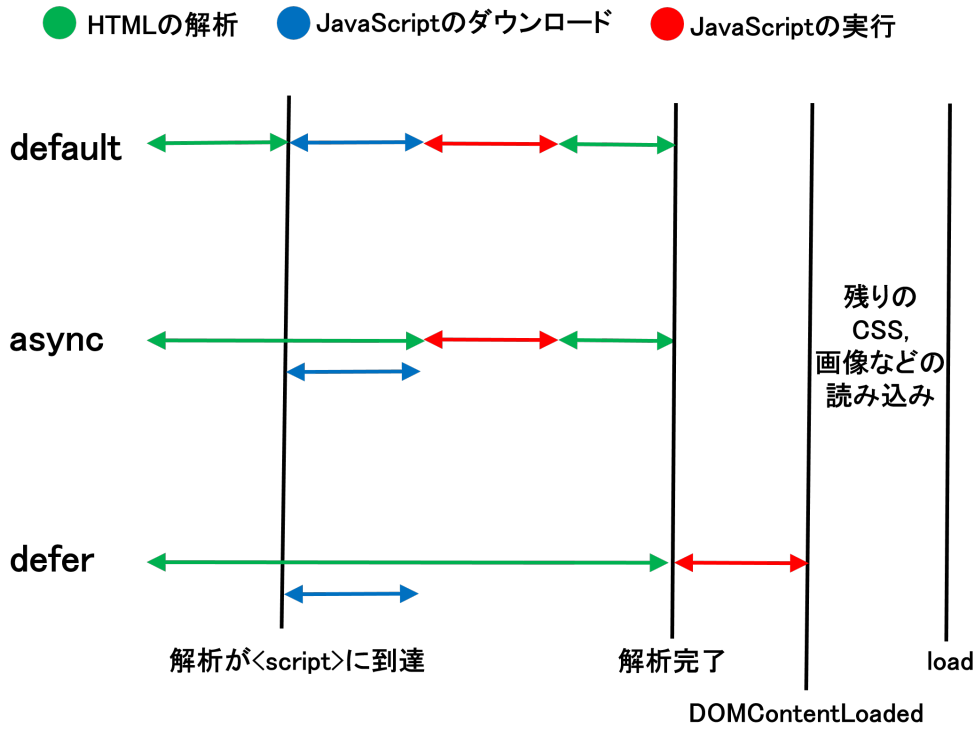

図 2.5 async/deferのダウンロードと実行のタイミング

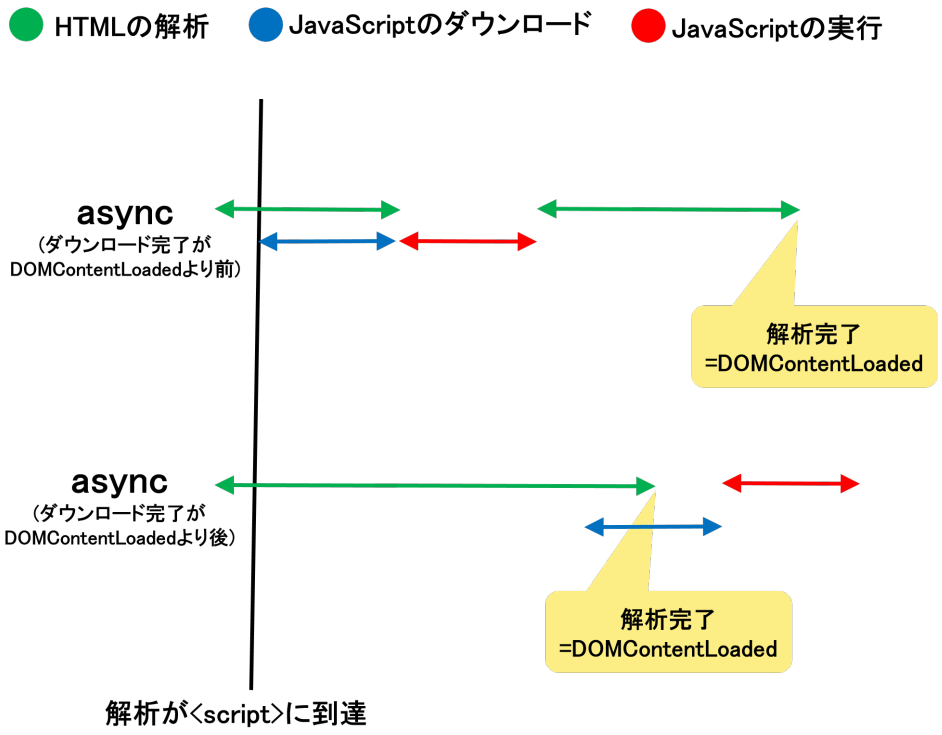

図 2.6 async:ダウンロード，実行タイミング

2.1.5 ファイルの参照

ブラウザはHTMLを解析し，HTMLに参照する様に書かれているファイルの要求を出す．その後受け取ったファイルを解析し，その結果参照するファイルがあればそのファイルの要求を出す．他のファイルを参照するファイルは，HTMLの他にCSS，JavaScriptなどがある．

2.1.6 通信特性制御のためのtcコマンド

tc（Traffic Control）コマンドはLinux Kernel内の通信を制御するコマンドである．tcコマンドはqdisc（Queueing Discipline）に対して操作を行う．図2.7に示すようにqdiscはnetwork device driverの上にある．本研究では帯域の制限と通信に遅延を発生させるために使用した．

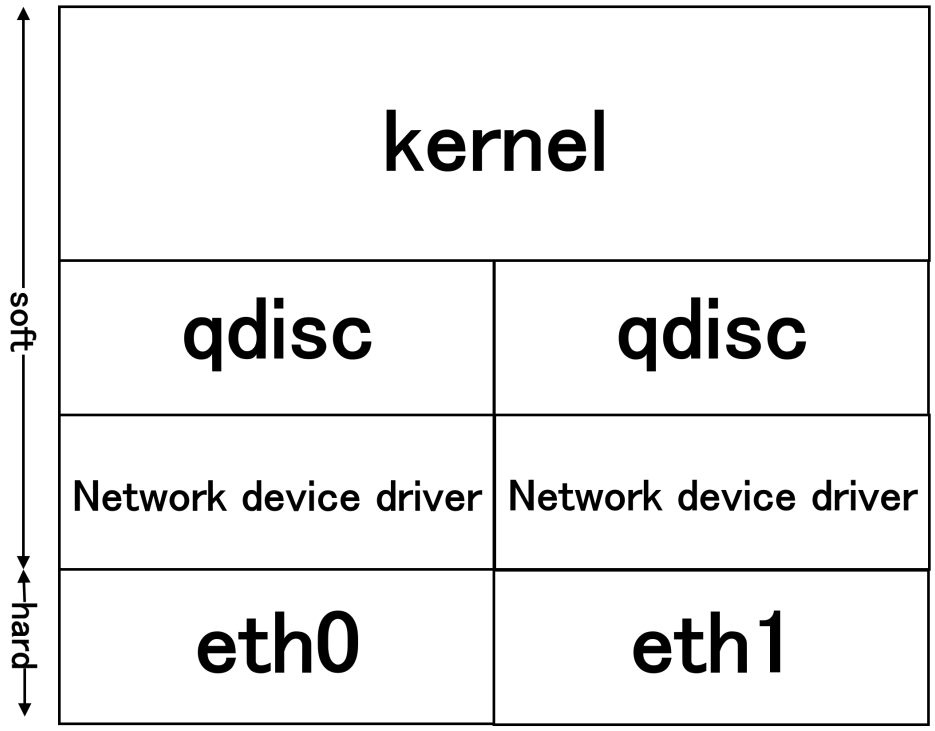

図 2.7 qdiscを含むデータ構造

2.1.7 表示速度の指数

Webページの表示速度を表す指標には種類が複数存在する．指標の名前とその定義を表2.1に示す．一般には，何らかの描画が始まる時点であるFirst Paintあるい

はFirst Meaning Paintが重要であるとされている．しかし，First Meaning Paintは機械的に測定することは困難である．このため本研究ではFirst Paintで評価するものとし，あわせて機械的に測定可能なPage Load Timeでも評価する．

表 2.1 Webページの表示速度を表す指標とその定義

First Paint	何かしらが描画されるまでの時間
First Meaning Paint	ユーザにとって意味があるものが描画されるまでの時間
Speed Index	経過時間による描画割合
Page Load Time	描画が完了するまでの時間

2.2 先行研究

2.2.1 サーバプッシュの効果的な利用

Webパフォーマンスの向上を目的としたサーバプッシュの研究[4]では，何を，いつどのような優先順位でプッシュすべきかに着目している．プッシュする優先順位は，データの種類で決定している．Webページを構成するファイルの内，CSS，JavaScript，画像の順でプッシュすることを提案している．しかし，この優先順位が全てのWebページにおいて適切とは限らない．例えば，先行研究では全てのCSSを最優先でプッシュするとしているが，2.1.3で示したように描画には必要のないCSSが存在する．以上のことからプッシュの優先順位には改善の余地がある.（技術課題 1）

また，文献[4]ではプッシュを止めるタイミングを重視している．多くのWebページ，例えば電子商取引では，購入履歴などの情報を含むページを動的に作成するが，これには時間を要する．ソケットに書き込まれたデータは取り消すことができないため，HTML作成中に行ったプッシュが終了するまで，HTMLを送信することができない．そのため，文献[4]では，HTMLのリクエストを受信してから，サーバでHTMLの作成が完了し送信可能となるまでの間だけプッシュすることを提案している．しかし，この手法では，図2.8に示す様に画面の表示に複数のデータが必要だった場合，その一部しか送られない可能性がある．その場合，そのデータを要求し，受け取るまで画面の描画を始めることができない．

また，図2.9に示す様に属性が追加されていないJavaScriptをプッシュできなかっ

た場合，2.1.4で示した様にそのJavaScriptの実行が完了するまでHTMLの解析は再開されない．そのため，JavaScriptとその<script>より下に記述されたファイルを2回に分けて要求しなければならない．以上のことからプッシュを止めるタイミングには改善の余地がある（技術課題 2）

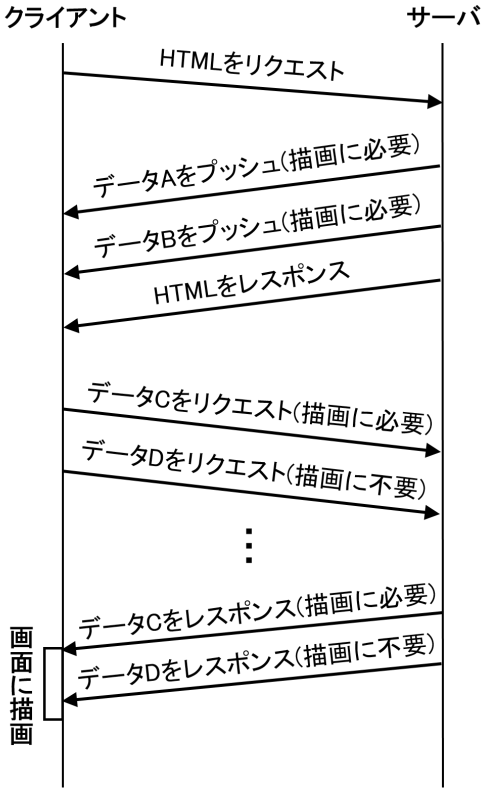

図 2.8 描画に必要なデータをプッシュしきれない場合のデータフロー図

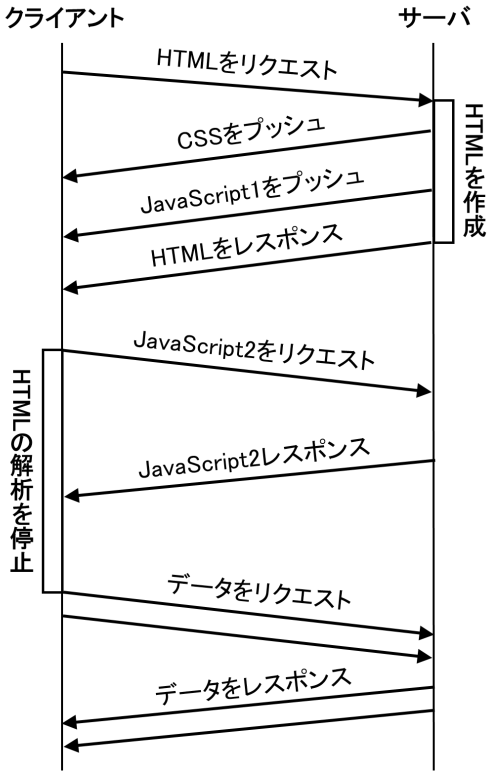

図 2.9 JavaScriptをプッシュしきれない場合のデータフロー図

2.2.2 コネクション集約とクロスオリジンサーバプッシュの実現

文献[5]ではプロキシによりコネクションを集約している．プロキシにデータを保存することで，HTMLのリクエストを受信した場合，そのWebページを表示させるのに必要なファイルをプロキシが全てプッシュすることを提案している．これにより，サーバとクライアントの往復通信回数は1回で済む，しかし，2.2.1で示した様に，ソケットに書き込まれたデータは取り消すことができない．そのため，画面の描画が開始されるのはそのWebページを構成するファイルが全て揃ってからになってしまう．適切な個所でプッシュを止めることが求められる,（技術課題 2）

3 提案

本章ではブラウザがコンテンツに対して設定した優先度（weightの値）を基にしてプッシュを制御する手法を提案する．

3.1 提案手法の概要

3.1.1 提案手法の1概要

技術課題1を解決するため，本研究ではブラウザが他のCSSより優先と判断したCSSと他のJavaScriptより優先と判断したJavaScriptを優先してプッシュすることを提案する．文献[4]とのデータフローの比較を図3.1に示す．

2.1.3で示した様にブラウザはメディアクエリの記述を基に，一部のCSSの優先度を他のCSSよりも低く設定する．加えて，2.1.5で示した様に最初に要求されるCSS群が他のCSSを要求することがある．その場合，図3.2の例で示す様にブラウザは最初に受け取ったCSSを解析するまでそのCSSを要求することができない．しかし，weightの値を元にプッシュすることで，そのようなCSSが描画に必要だった場合でも，優先してプッシュすることができる．これにより，図3.1に示す様に,描画に必要なCSSが解析前にブラウザ内に揃うことになる．

JavaScriptについては，2.1.4で示した様にブラウザは，<script>タグに追加された属性を基に一部のJavaScriptの優先度を他のJSよりも低く設定する．加えて，2.1.5で示した様に最初に要求されるJS群が他のCSSやJavaScriptを要求することがある．その場合も，weightの値を元にプッシュすることで，そのようなCSSやJavaScriptが描画に必要だった場合でも，優先してプッシュすることができる．

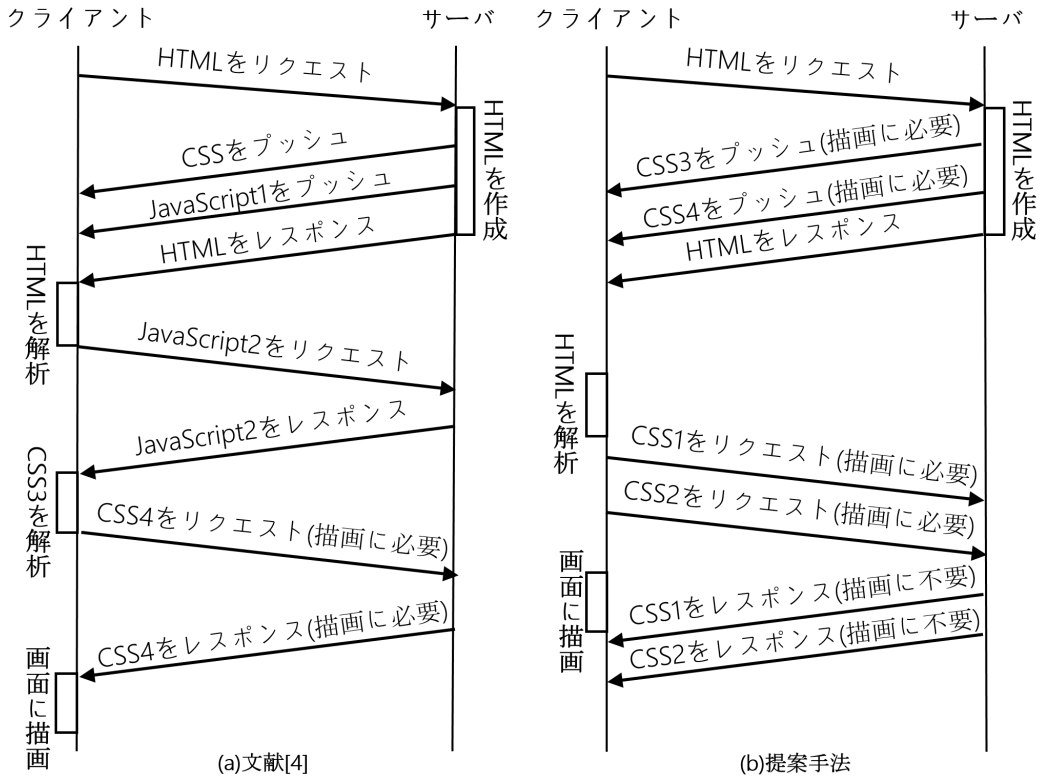

図 3.1 描画に必要なCSSが他のCSSから参照されていた場合の文献[4]と提案手法の比較

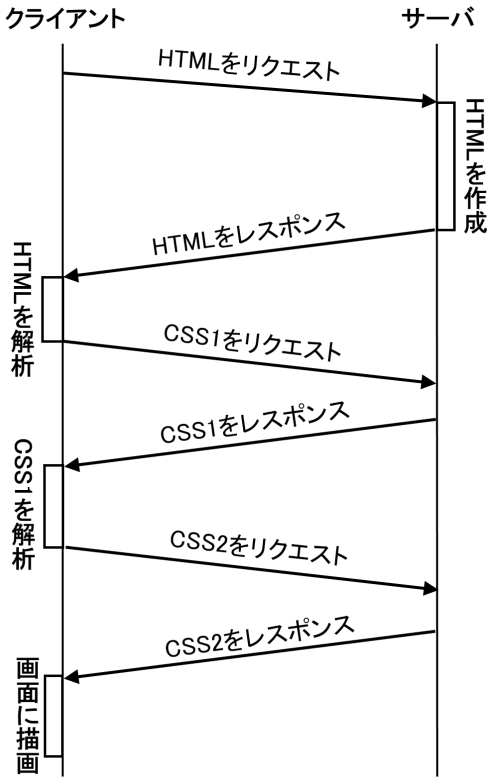

図 3.2 CSSが他のCSSから参照されていた場合のデータフロー図

3.1.2 提案手法の2概要

技術課題2を解決するため，優先度の高いCSSと優先度の高いJavaScriptは，HTML作成時間を超えてでも全てプッシュしてからHTMLを送信することを提案する．文献[4]，[5]それぞれとのデータフローの比較を図3.3, 3.4に示す.

現在のブラウザは必要なCSSとJavaScriptが揃わないと描画を開始しない場合が多い．加えて2.1.4で示した様に，属性がないJavaScriptのダウンロードと実行を行なっている間は，HTMLの解析を中断しなければならない．HTMLの解析の中断により，それらのJavaScriptより下に記述されたファイルは，Javascriptの実行が完了してから要求を出さなければならない．しかし，優先度の高いJavaScriptを全てプッシュすることで，ダウンロードによるHTMLの解析の中断を防ぐことができる．これにより，HTMLの作成時間を超えてプッシュした時間より，JavaScriptのダウンロードにかかる時間が長い場合，[4]と比較して，より早く画面表示ができる．

また，文献[5]では全てのファイルプッシュしているが，プッシュする全てのファイルが描画に必要だとは限らない．描画に必要なファイルをweightの値から判断し

てプッシュすることで，描画に必要なファイルが全て揃った状態でかつ，より早くHTMLを受け取ることができる．そのため，先行研究と比較してより早く描画が開始できる．

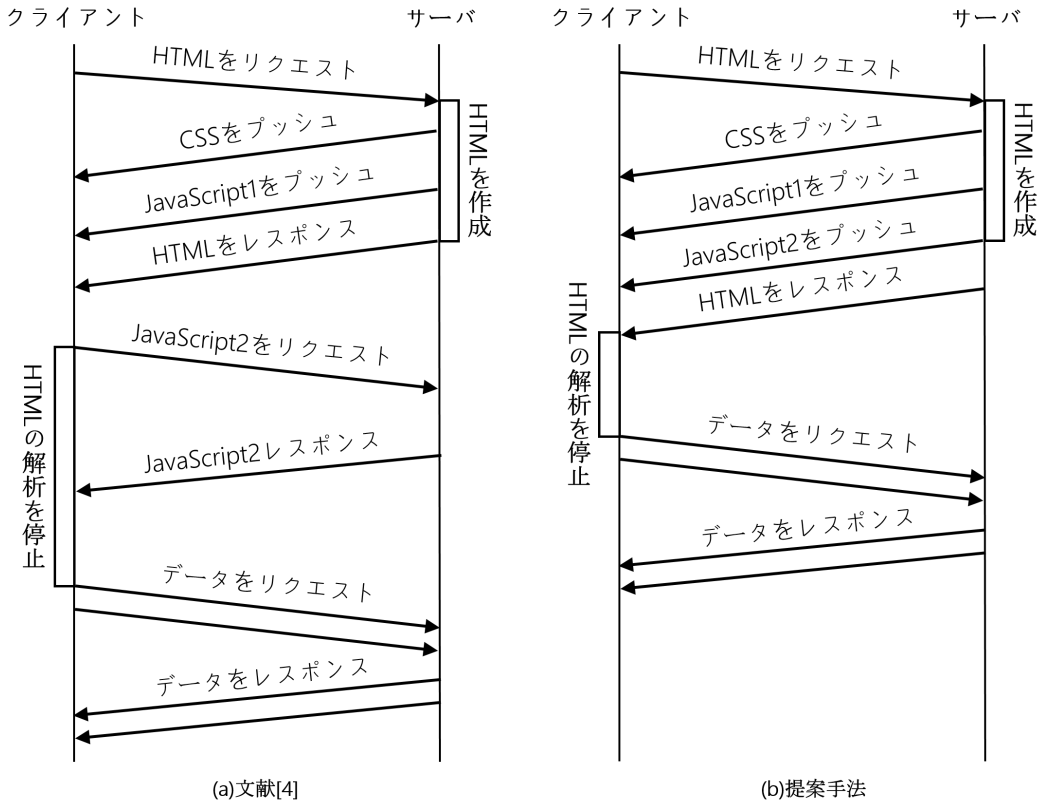

図 3.3 属性がないJavaScriptをプッシュできなかった場合の文献[4]と提案手法の比較

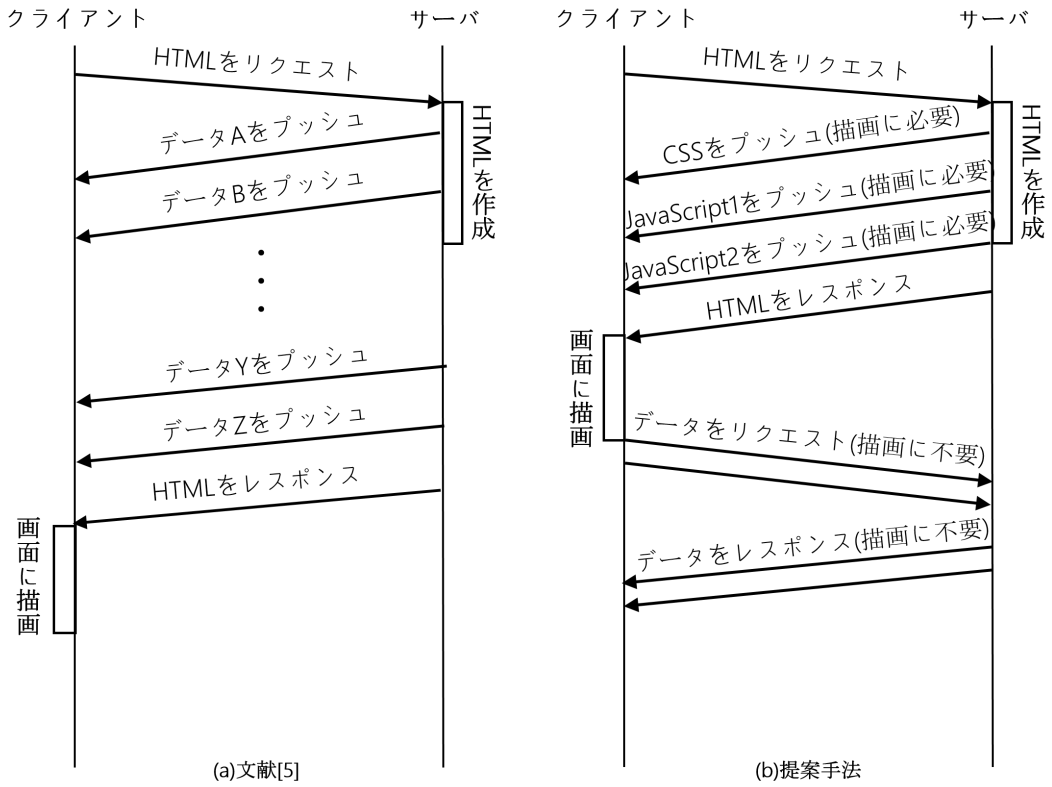

図 3.4 文献[5]と提案手法の比較

4 実装

本章では開発環境と，前章で述べた提案手法を実現するためのシステムについて述べる．

4.1 開発環境

サーバ側，クライアント側の実装環境をそれぞれ表4.1，表4.2に示す.本研究ではサーバ側の実装にはNode.jsを用いた．Node.jsはサーバ側でJavaScriptを実行できる様にするプラットフォームである．

表 4.1 サーバの実装環境

OS	Ubuntu 18.04.2 LTS
CPU	Intel（R） Core（TM）i5-9600 CPU @ 3.10Gz
Node.js	Node.js 12.8.0

表 4.2 クライアントの実装環境

OS	Ubuntu 18.04.2 LTS
CPU	Intel（R） Core（TM）i7 CPU 870 @ 2.93Gz
GoogleChrome	Chrome 79.0.3945.117

4.2 システムの概要

本節では実装したシステムについて述べる．本研究におけるサーバのデータ構造を図4.1に，システムのデータフローを図4.2に示す.

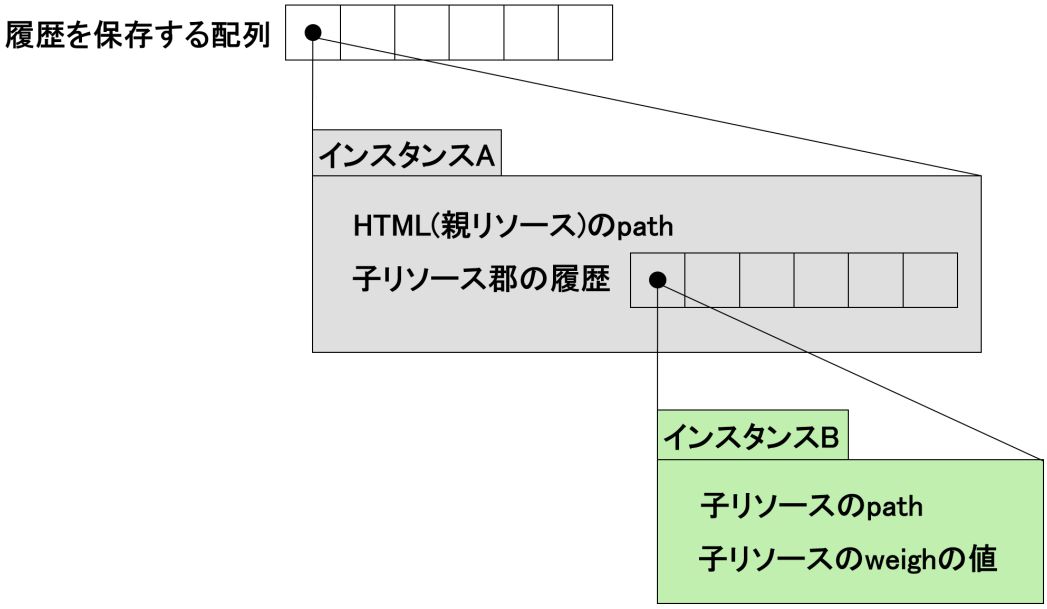

図 4.1 サーバのデータ構造

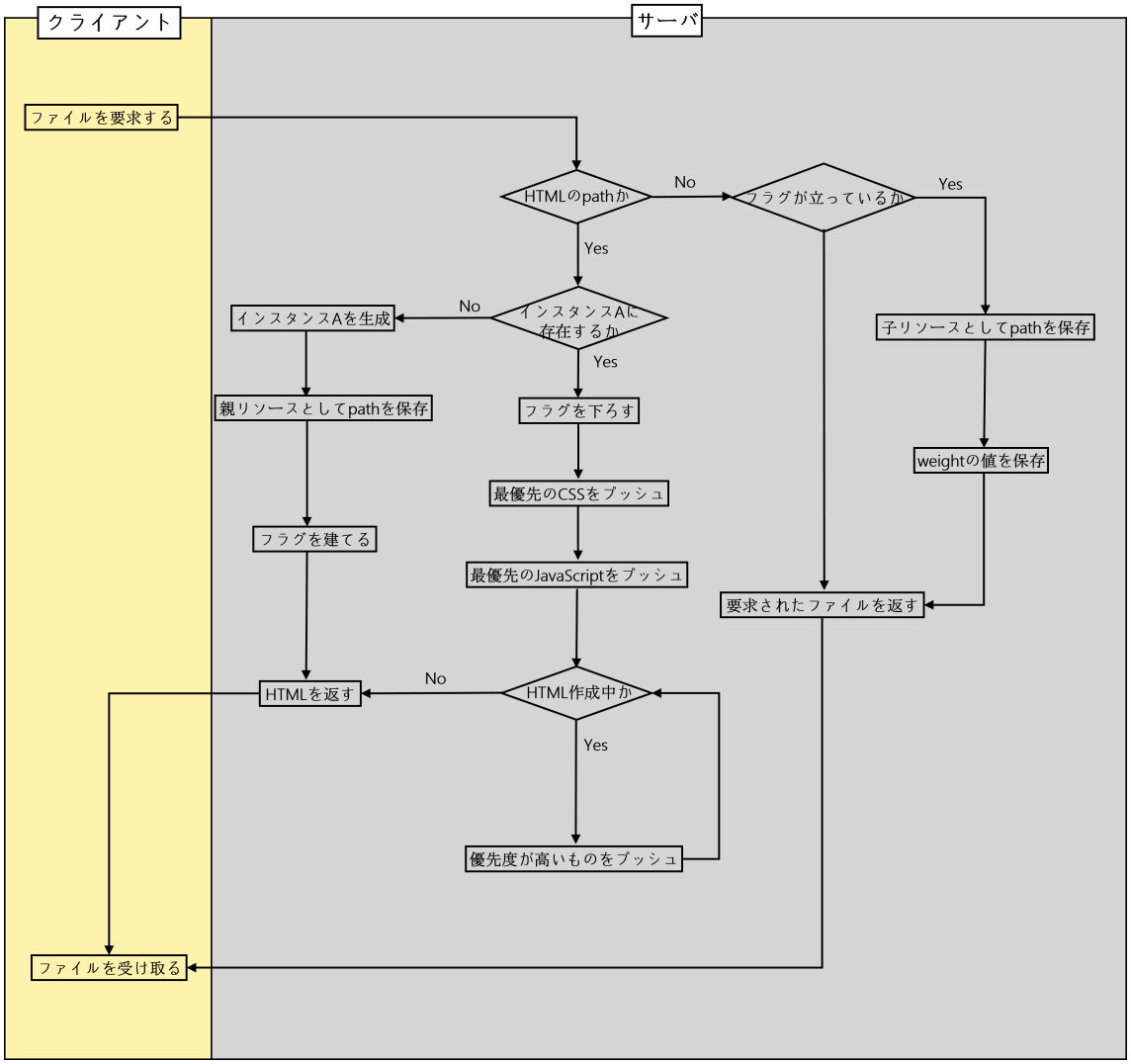

図 4.2 システムのデータフロー

4.2.1 要求履歴の保存

プッシュのためにはWebページの表示に必要なファイルを特定する必要がある．具体的には，HTMLの要求が来た際に，そのHTMLが参照するファイルをサーバ側は事前に把握している必要がある．これにはHTMLだけでなく，CSSなどが参照するファイルも含まれる．本研究ではHTMLの要求が来たことでWebページが遷移したと判断し，次のHTMLの要求が来るまでに要求されたファイルは最初に要求されたHTML（親リソースと呼ぶ）のの表示のために必要なファイル群（子リソースと呼ぶ）とみなす.

サーバはクライアントから要求が来た場合，headerのpathの値を見る．pathの値はクライアントが要求しているファイルの場所と名前を表している．この値を元に，クライアントが要求しているファイルがHTMLなのか，それ以外のファイルなのかを判断する．要求されたファイルがHTMLだった場合，そのHTMLのpathが履歴に保存されているかを確認する．保存されていれば，履歴を元にプッシュする．履歴に保存されていない場合，インスタンスを生成し，HTMLのpathを保存する．その際，フラグを立てる．その後要求されるファイルは子リソースとして，pathとweightの値を親ソースと関連付けて保存する．以上の処理を履歴に存在するHTMLの要求が来て，フラグを下ろすまで行う.

4.2.2 プッシュする順序の決定

図4.2はデータフローを表しているため図中には明示していないが，HTMLを返した後に子リソースの履歴のソートを行なっている．これはプッシュの度に履歴を全て見るのを防ぐためである．ソートは“最優先のCSS，最優先のJavaScript，それ以外の優先度の降順”になる様に行なっており，これがそのままプッシュする順番となっている．ただし単に優先度の降順にソートすると，フォントが最優先のCSSと同等の順となってしまう．フォントは描画を開始するに当たっては必要ないため，最優先のCSSと最優先のJavaScriptの後になる様にソートしている．なお，優先度が同じ場合は要求された順番を保つ様にソートしている.

4.2.3 プッシュ方法

プッシュする際は，まず最優先のCSSと最優先のJavaScriptをプッシュする．その後，HTMLの作成が完了していなければ，HTMLの作成中にプッシュできる範囲でプッシュを行う．

HTMLを作成中のみという決まった時間内にプッシュできるデータ量を知るためには，サーバはクライアント間との帯域を把握している必要がある．これは本来の実装であれば，既存技術を用いて動的に測定すべきであるが，今回の実装ではサーバとクライアント間の帯域を事前に測定することで対処を行った．測定した帯域からHTMLの作成中にプッシュできる容量を事前に計算し，サーバに登録している．プッシュする前にファイルのサイズを取得し，プッシュした合計のファイルサイズとプッシュできる容量を比較することでプッシュの可否を判断する．

また，本来であればHTMLの作成に要する時間は実行時に測定すべきであるが，今回の実装ではHTMLの作成を実際には行わず，要求されてからHTMLを遅れて返すことで，HTMLをサーバ側で作成したと仮定している．この実装にはsetTimeOut関数を用いた．setTimeOutは第一引数で指定した時間分だけ，第二引数の関数を遅らせて実行することができる．なお，プッシュを行う場合はHTMLが最後に送信される様に，プッシュが完了次第HTMLを返している．

PUSH_PROMISEを送信したにも関わらずそのファイルの要求が来た場合，本来はそのファイルのプッシュはやめるべきだが，現在の実装では，プッシュをプッシュとレスポンスの両方を行う．PUSH_PROMISEの送信前にそのファイルの要求が来た場合も，プッシュとレスポンスの両方を行う．

5 評価

クライアントのブラウザにはChromeを使用した．サーバとクライアントの帯域制限には，tcコマンドを用いて仮想的なネットワーク環境を構築した．文献[4]，[5]の提案手法と本研究の提案手法を実装し，2つのコンテンツでそれぞれFirst Paint，Page Load Timeを5回測定し平均をとる．First PaintとPage Load TimeはChromeの拡張機能を用いて測定した．レイテンシを200ms，HTMLの作成時間を10msとした．キャッシュは実験結果に大きく影響を与えるため，キャッシュが残らない様に2つの方法を用いて削除を行った．以下に実験の手順を示す．

ブラウザを立ち上げたのち，Chromeの拡張機能を用いてキャッシュの削除を行う．削除が終了したのち，サーバにアクセスしFirst PaintとPage Load Timeの測定をする．測定後ブラウザを閉じ，Linux上でキャッシュのファイルを直接削除する．これを1セットとし，これを5回行った．ただし，事前にサーバに複数回アクセスすることで，サーバ側に履歴は作成済みである．

5.1 コンテンツ

測定は2つのコンテンツで行った．評価対象のコンテンツAは，優先度が高いCSSとJavaScriptのみから構成され，あるCSSが他のCSSを参照している．HTMLが1個，CSSが4個，JavaScriptが5個，フォントが9個から構成されている．このコンテンツAは実際のWebページをベースにしており，一般的なコンテンツの構成である．もう1つのコンテンツBは，優先度が低いCSSとJavaScriptが含まれ，JavaScriptがCSSを参照している．HTMLが1個，CSSが8個，JavaScriptが14個，画像が54個，フォントが3個から構成されている．このコンテンツBは，2.1.4で示した機能の使用が今後増えることを想定して，asyncの属性を持つJavaScriptを含んでいる．

5.2 コンテンツAの実験結果と考察

実験結果を図5.1に示す．コンテンツAの実験では，First Paintは通常のサーバと文献[5]の2つと比較して提案手法の方が早かった．これは描画に必要な子リソースをプッシュしたことにより，描画を始めるまでの往復通信回数が減少したためと考えられる．一方，文献[4]とは同等であった．これはHTML作成中にどちらも描画に

必要な子リソースをプッシュできたためと考えられる．また，Page Load Timeについては文献[5]が最も早かった．これは全ての子リソースをプッシュしたことにより，サーバとクライアント間の往復通信回数が1回のみだったためだと考えられる．

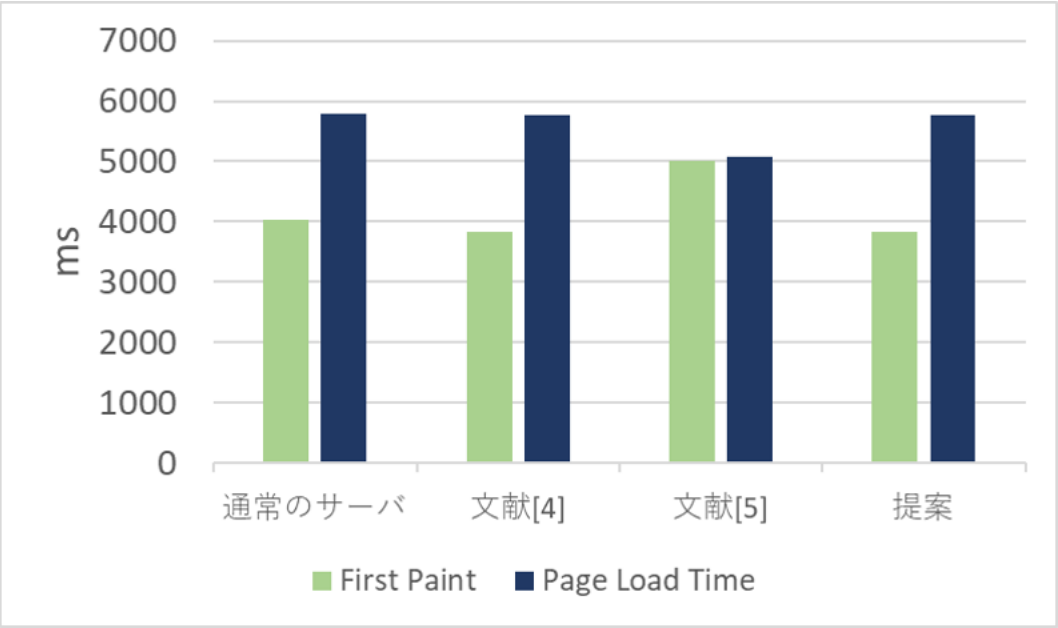

図 5.1 コンテンツAの実験結果

5.3 コンテンツBの実験結果と考察

実験結果を図5.2に示す．コンテンツBでは，First Paintは文献[4]や[5]よりも高速であったが，通常のサーバとは同等であった．原因を特定するためにサーバ側でログをとった．通常のサーバ，文献[4], 提案手法それぞれのログの出力結果を図5.3, 5.4, 5.5に示す．

図5.5からプッシュを開始してから，プッシュが終わるまでの時間が800ms程度であることがわかる．プッシュのログはプッシュのコールバックで出力している．ログが出されるタイミングはプッシュするファイルのデータが，ソケットに書き込まれたタイミングである．このことからファイルの読み込みに800ms程度を要していることが推定される．図5.4でも同様にファイルの読み込みに800ms程度を要していると考えられる．さらに，図5.3と5.4から，どちらの場合でもCSSとJavaScriptの要求の合計数が共に10個になると，次の要求が来るまで400ms程度を要しているこ

とがわかる．これは，今回利用したブラウザ（Google Chrome 79.0.3945.117）の実装においては同時にダウンロードできるファイル数が10個であり，JavaScriptの実行が完了するまで次の要求が出せなかったという可能性が考えられる．図5.5から提案手法では属性のないJavaScriptを全てプッシュしているため，この制限による往復通信回数の増加の影響を受けなかったと考えられる．図5.3と5.4からどちらも2回に分けて画像群の要求がされていることがわかる．一方で．図5.5から提案手法では画像群の要求が分かれることなく行われていることがわかる．以上のことから，描画には画像が必要であり，プッシュするファイルの読み込みに800ms要したため，往復通信回数の2回減少の効果が隠蔽されてしまったと考えられる．

また，Page Load Timeはコンテンツと同様の理由で文献[5]が最も早かった．提案手法と通常のサーバを比較した場合，First Paintと同様の理由により大きな差がなかったと考えられる．文献[4]が遅かった理由は，プッシュによる通信回数を減らせなかったことに加え，プッシュするファイル読み込みに時間を要したためと考えられる．以上より，往復通信回数を減らすことはできたが，ファイルの読み込み時間により提案手法では大きな効果が得られなかったと考えられる．

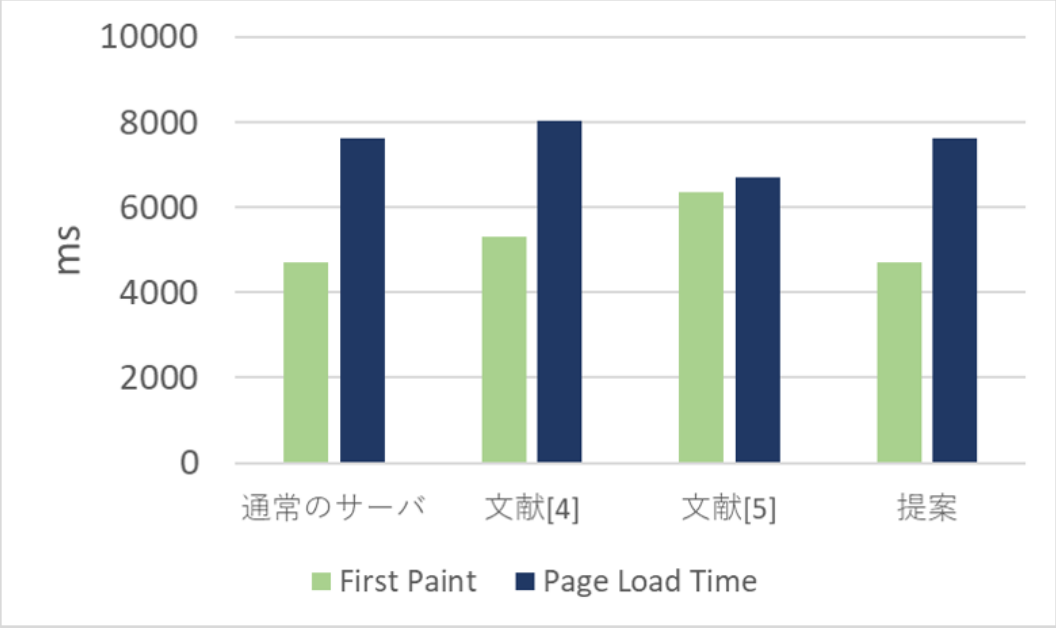

図 5.2 コンテンツBの実験結果

```
HTMLの要求を受信
//////////////////約855ms/////////////
CSS郡の要求(4個)
JavaScript郡の要求(6個)
//////////////////約396ms/////////////
JavaScript郡の要求(7個)
JavaScript(async)
画像の要求
//////////////////約1198ms/////////////
画像郡の要求
//////////////////約393ms/////////////
画像郡の要求
描画に不要なCSS郡の要求
画像郡の要求
フォントのを参照するCSSの要求
//////////////////約547ms/////////////
フォント(woff2)の要求
//////////////////約848ms/////////////
フォント(woff)の要求
//////////////////約240ms/////////////
フォント(otf)の要求
```

図 5.3 通常のサーバのログの出力結果

```
HTMLの要求
//////////////////約1201ms/////////////
プッシュを開始
//////////////////約803ms/////////////
プッシュ終了
//////////////////約454ms/////////////
JavaScript郡の要求(10個)
//////////////////約395ms/////////////
JavaScript郡の要求(3個)
JavaScriptの要求(async)
画像の要求
//////////////////約416ms/////////////
画像郡の要求
//////////////////約379ms/////////////
画像郡の要求
//////////////////約212ms/////////////
フォント(woff2)の要求
//////////////////約849ms/////////////
フォント(woff)の要求
//////////////////約410ms/////////////
フォント(otf)の要求
```

図 5.4 文献[4]のログの出力結果

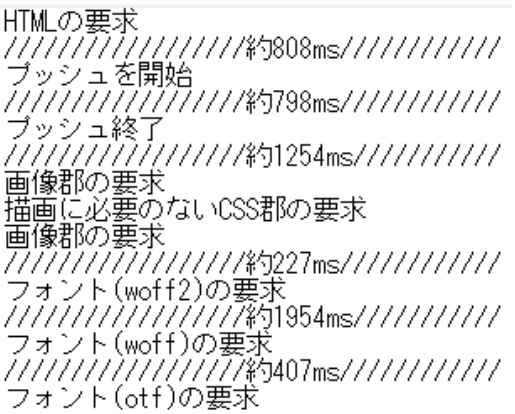

図 5.5 提案手法のログの出力結果

5.4 実験結果のまとめ

コンテンツAの実験では，First Paintは通常のサーバと文献[5]の2つと比較して提案手法の方が早かった．一方，文献[4]とは同等であった．また，Page Load Timeについては文献[5]が最も早かった．

コンテンツBの実験では，First Paintは通常のサーバと同等であったが，文献[4], [5]の2つと比較して早かった．また，Page Load Timeについては文献[5]が最も早く，提案方式は通常のサーバとは同等，文献[4]よりは高速であった．

以上のことから，閲覧時に特に重要なFirst Paintについては既存技術と同等以上であることが確認でき，本提案方式の有効性がある程度示せた．

6 おわりに

本稿ではまず，描画に要する時間が閲覧者に与える影響について述べ，表示を高速にすることの重要性について述べた．次に，先行研究の概要とその問題点を指摘した．これらの問題を解決するために，描画に必要のないファイルを判断し．描画に必要なファイルを優先してプッシュする方式を提案した．本方式では，ブラウザが設定したファイルの優先度を記録しておき，次に同じ要求が来た際にはこの情報を利用してプッシュする．特に最優先のCSSとJavaScriptについては，要求されたHTMLの送信を遅らせてでもプッシュすることで，表示の高速化を実現する．次に提案方式を実装し評価を行った．First Paintにおいては通常のサーバ，文献[4]および文献[5]と同等または早いことが確認できた．Page Load Timeについては，通常のサーバ，文献[4]と同等または早いことが確認できた．しかし，サーバログから往復通信回数を減らすことはできたが，ファイルの読み込み時間により提案手法では大きな効果が得られなかったと考えられる．ファイル読み込みに時間を要したことにより，往復通信回数の減少の効果が隠蔽されてしまったと考えられる．ファイルの読み込みを高速化する手法は今後の課題である．

参考文献

[1] Y. Takahashi, ”0.1秒の遅れが、1%の売上に影響する!? ページ表示速度の影響力と改善法まとめ,” 3MEDIA, http://media.marsdesign.co.jp/web/how-to-check-and-improve-web-performance.html.

[2] Medium, ”Impact of slow page load time on website performance,” Get smarter about what matters to you., https://medium.com/@vikigreen/impact-of-slow-page-load-time-on-website-performance-40d5c9ce568a.

[3] Caddy, ”Server Push is Hard,” The Caddy Blog, https://caddyserver.com/blog/implementing-http2-isnt-trivial.

[4] K. Zarifis, M. Holland, ManishJain, E. KatzBassett, and R. Govindan, “Making Effective Use of HTTP/2 Server Push in Content Delivery Networks”, Technical Report 17-971, Univ of Southern California, 2019.

[5] K. Sawada, Y. Kitaguchi, K. Yamaoka. “Cross-Orizin Server Pushing over Aggregated Connection for Fast Web Distribution,” IPSJ SIG Technical Report, Vol.2019-IOT-44, No.26, pp. 1–8, 2019.

[6] M. Belshe, BitGo, R. Peon, Google, Inc, M. Thomson, Ed., Mozilla, ”Hypertext Transfer Protocol Version 2 (HTTP/2),” Internet Engineering Task Force(IETF), http://www.rfc-editor.org/rfc/rfc7540.txt.

[7] Mozilla Developer Network(MDN), ”CSS basics,” MDN web docs moz://a, https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/CSS_basics.

[8] Mozilla Developer Network(MDN), ”Using media queries,” MDN web docs moz://a, https://developer.mozilla.org/en-US/docs/Web/CSS/Media_Queries/Using_media_queries.

[9] Mozilla Developer Network(MDN), ”Window: DOMContentLoaded event,” MDN web docs moz://a, https://developer.mozilla.org/en-US/docs/Web/API/Window/DOMContentLoaded_event.

[10] Mozilla Developer Network(MDN), "Window: load event,"" MDN web docs moz://a, https://developer.mozilla.org/en-US/docs/Web/API/Window/load_event.

[11] Mozilla Developer Network(MDN), "What is JavaScript?," MDN web docs moz://a, https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript

[12] Web Hypertext Application Technology Working Group（WHATWG), "4.12 Scripting," HTML, https://html.spec.whatwg.org/multipage/scripting.html#attr-script-async.

[13] Web Hypertext Application Technology Working Group（WHATWG), "12.2.7 The end," HTML, https://html.spec.whatwg.org/multipage/parsing.html#the-end.