RO-009

サーバサイドイメージレンダリングによる Web UI コンポーネントフレームワークの試作と評価
Prototyping and Evaluation for Web User Interface Component Framework using Server Side Image Rendering

内海 宏律† Hironori Utsumi　黄 双全† Shuangquan Huang　菊地 大介† Daisuke Kikuchi　齋藤 邦夫† Kunio Saito　手塚 大† Masaru Tezuka

１．はじめに

スマートデバイスが広く普及し，既存の業務システムを Web 化しスマートデバイスの Web ブラウザから利用するニーズが増加している。また，クラウドファースト，モバイルファーストが提唱され，スマートデバイスへの対応は必須の要件となっている。Web アプリケーションのユーザインタフェース(UI)はボタンなどのコンポーネントと呼ばれるソフトウェア部品で構築されている。コンポーネントはボタンやテキストボックスなどの単機能なものだけではなく，グラフや表計算，ネットワーク図など複雑な表示や操作が可能なものもある。これらの複雑なコンポーネントはデータのレンダリングに多くの計算量を必要としたり画面表示に多くのデータ量を必要とするなどの特徴がある。スマートデバイスは PC と比較すると性能が低く，このような特徴を持つコンポーネントを利用すると性能問題が発生する。また，3G 回線など不安定な通信環境下で利用されることも多く，データ通信に伴う待ち時間も長くなる。そこで，このような課題を解決し，スマートデバイス上で動作するコンポーネントを開発するための Web UI コンポーネントフレームワークを提案する。

２．スマートデバイスで負荷の高いコンポーネントを利用する際の課題

Web アプリケーションでは Model-View-Controller(MVC) アーキテクチャを適用し，図 1 のような構成をとることが多い。

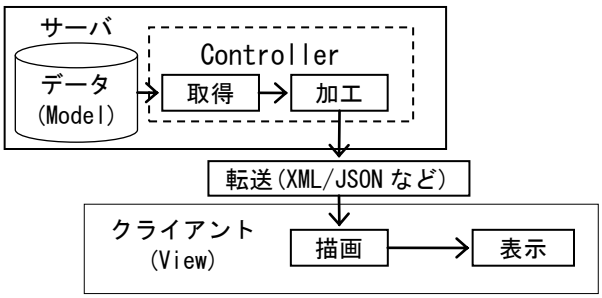

図 1 Web システムの処理フロー

クラウド上のサーバに集約されたデータを Model とし，View であるクライアントの要求に応じてサーバ上の Controller がデータの取得および加工を行う。その結果を XML や JSON などにシリアライズ化し，クライアントに転送する。クライアントではその結果を画面上に描画し表示する。PC での利用を前提とした従来の Web アプリケーションで用いられるコンポーネントには ActiveX，Microsoft Silverlight，Adobe Flash や Java アプレットが用いられる。これらのコンポーネントをスマートデバイス上で実行した場合，処理負荷が高く UI 操作に対する応答に時間がかかり十分な操作性が確保できない。また，タッチスクリーンでの操作に対応していないものや特定のプラットフォーム専用であるものも多くマルチデバイスを謳う Web アプリケーションへの適用は難しい。マルチプラットフォームに対応した JavaScript による Document Object Model (DOM)操作や HTML5 Canvas を利用して画面を描画する方法もあるが，処理性能が低いスマートデバイスでは負荷が高く，画面表示までに長時間かかる場合やスクロール時の再描画処理で CPU が占有され UI 操作に対する応答が遅延する問題がある。

Canvas による描画では，画面を複数のレイヤーに分割し，再描画対象とするオブジェクトを限定することで高速化する手法が提案されている[1]。この手法はゲーム画面でのキャラクター画像などのように画面の一部を頻繁に更新する際に特に有効となる。しかし，画面の初期表示や画面全体のスクロールに伴う再描画処理では全レイヤーが描画対象となるため大きな効果は得られない。さらに，レイヤー数の増加に伴いクライアントのメモリ使用量も増加するトレードオフの関係があるため，リソースに制限のあるスマートデバイスでは適用しにくい。

また，画面の表示に多くのデータ量を必要とするコンポーネントの場合，不安定な通信環境下で画面表示までの時間を短縮するためには，データを一括転送せずに，表示に必要なデータのみを分別し転送する差分転送が有効となる。しかし，転送すべきデータの分別はデータモデルの構造やレンダリングアルゴリズムなどに強く依存し，汎用的な処理は容易ではない。結果として，データ全体の一括転送が必要となり画面表示までの待ち時間が長くなる。以上よりスマートデバイス上で従来方式のコンポーネントを利用する場合の課題を下記の通り課題(1)～(4)として整理した。

課題(1) マルチデバイス，マルチプラットフォームで広く利用できない
課題(2) 処理負荷が高く十分な応答性能が得られない
課題(3) 描画に必要なデータ量が多く待ち時間が長くなる
課題(4) 転送すべきデータの分別はデータモデルの構造やレンダリングアルゴリズムに強く依存するため汎用的な処理が困難

†（株）日立ソリューションズ東日本, Hitachi Solutions East Japan, Ltd.

3. サーバサイドイメージレンダリングを活用したWeb UIコンポーネント

3.1 サーバサイドイメージレンダリングの提案

3.1.1 サーバサイドイメージレンダリングの概要

従来手法では処理負荷の高い描画をクライアント上で行っていたため応答性が低下する原因となっていた。また，汎用的な差分転送が困難であるためデータを一括転送する必要があり，待ち時間が長くなっていた。そこで，図 2 に示すように描画をサーバ上で実行するサーバサイドイメージレンダリング方式を提案し課題の解決を図った[2]。この方式では多くの計算量を必要とする描画処理をサーバ上で行うことで，クライアントの負荷を下げることができ，操作性能を確保できる。また，描画結果は画像としてクライアントに転送するため，クライアントの画面表示に必要なデータを画像の座標情報のみで容易に分別可能となる。初期表示時は表示に必要な領域の画像データのみ転送することで待ち時間を短縮することができる。残りのデータはユーザのスクロール操作などにあわせて差分転送する。高速なサーバ上での描画処理と画像の差分転送を組み合わせることで，初期表示までの時間を短縮し，UI 操作に対する応答性能も確保可能となる。

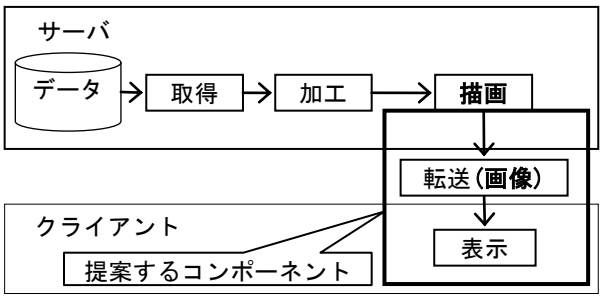

図 2 サーバ上の描画結果を表示するコンポーネント

ガントチャートと呼ばれるグラフを表示するコンポーネントを題材として予備実験を行った[2]。提案手法は表 1 に示す通り Canvas を利用する方法と比較し約 60%程度の時間で描画処理を完了することができた。また，画面の初期表示までに必要なデータサイズを比較したところ，表 2 に示す通り XML 形式にシリアライズする従来方式と比較し，約 10%程度のデータ量で画面の表示が可能という結果が得られており，課題(3)が解決できる。さらに転送すべきデータの分別は画面表示に必要となる領域の座標情報のみで容易に分別可能となるため，課題(4)が容易に解決できる。

表 1 描画時間

Canvas 描画方式	238ms
サーバサイドレンダリング方式	138ms

表 2 転送されるデータサイズ

XML データ	664kB
PNG 画像(初期表示領域のみ)	56kB

クライアントはサーバから受信した画像を表示するのみとなるため，多くの計算量を必要とする処理は不要となる。また，スクロール処理は既にレンダリングが完了した画像の表示位置を変更するのみで実現可能となり，Canvas を利用した場合のように負荷の高い再描画処理を実行する必要がなくなりクライアントの応答性能が向上する。これにより，課題(2)の解決を図ることができる。

実装面では，JavaScript による実装が可能でありマルチプラットフォームに対応したコンポーネントとすることができるため，従来方式のコンポーネントを利用する際に問題となる課題(1)が解決できる。

3.1.2 サーバサイドイメージレンダリングの課題

サーバサイドで画像の描画を行った場合，クライアントサイドで画像に対する書き換えなどの加工は難しくなる。アプリケーションに適用した際の機能要件を考慮すると以下の問題が想定できる。

(1) 画面スクロールの問題

スクロール時には 1 枚の画像全体がスクロールされるため，グラフ画面の目盛など画面内に残したい領域(固定領域)も画面外にスクロールされてしまう。

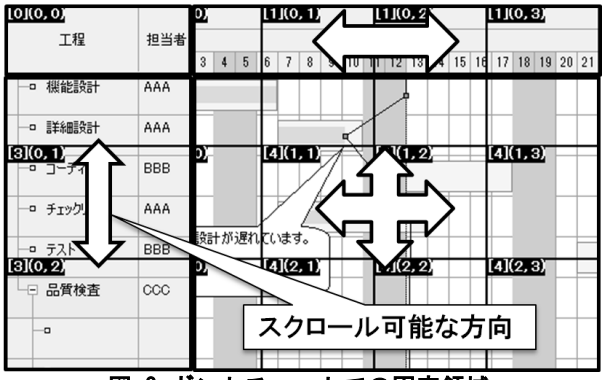

図 3 ガントチャートでの固定領域

例えば，図 3 に示すガントチャート画面では，左側に作業項目，上部にその作業を行う日程を示すカレンダーを表示する画面構成をとる。そして，作業項目は上下方向，カレンダーは左右方向のみにスクロールさせ，常に画面内に表示されるように制御する。このような構成を持つ画面に対し，画像全体が一律にスクロールすると，作業項目などの部分もそれにあわせて画面外へとスクロールされてしまい，内容が見にくくなってしまう。その結果，画面の端にある作業項目やカレンダーを確認するためには，画面全体を一度端までスクロールさせる必要がある。

(2) 追加描画に関する問題

経路案内アプリケーションでの地図上の経路表示やガントチャート上での選択範囲領域を示すラバーバンド(選択範囲を示す矩形)表示などのように，サーバで生成した画像上に何らかのオブジェクトを描画するニーズは多い。画像を利用する方式の場合，一度サーバに変更内容を通知し，サーバ上で再描画を行った結果をクライアントに転送し表示する方式では，通信帯域に制限のあるスマートデバイスでは画面更新までに時間がかかる。また，回線切断時に動作を継続することが困難となる。

3.1.3 固定領域の設定

Web ブラウザ上に配置した画像は画像全体が同じ方向にスクロールするため，その矩形内で異なった方向にスクロールさせることができない。そこで，画像がスクロ

ール方向の異なる境界を跨がないようにするために，サーバ上で固定領域ごとに分けて画像を生成することで固定領域を実現した。クライアント側のコンポーネントの上下に横方向だけにスクロールする領域と，左右に縦方向だけにスクロールする領域を用意し，クライアント画面の上下左右部分に常に画面内に表示可能な領域を設定可能とした。さらに，固定領域とする部分の組み合わせは任意に指定可能とし，アプリケーションの画面構成に柔軟に対応可能とした。また，各固定領域の画像は領域ごとにスクロール可能な方向に対してブロック状に分割することで，UI のスクロールに応じて画像を差分送信可能とした。

3.1.4 クライアント描画レイヤー

ラバーバンドや簡単な経路案内など，描画負荷が高くないものはサーバで描画せずにクライアント側で描画した方が応答時間の短縮につながる。そこで，クライアントサイドのみで完結する描画を可能とするため，サーバで生成した画像上に Canvas および div 領域を重ねて表示し，その上に JavaScript を利用して任意の画面描画を行うことができる構造とした。アプリケーションはこれらのレイヤー上に描画を行うことで，サーバで生成した画像上に上書きしたような表示が可能となる。また，画面のスクロール時には，コンポーネントがこれらのレイヤーも同時にスクロールすることで，レイヤー上の描画結果もサーバで生成した画像と同期してスクロールする。これによりアプリケーション側でのスクロール制御を不要とした。

3.1.5 差分転送

コンポーネントフレームワークは，描画結果を画像データとして処理するため，クライアント側のコンポーネントの画面表示に必要となるデータを座標情報のみで容易に分別でき，課題(4)が解決できる。つまり，表示するデータモデルの構造やレンダリングアルゴリズムによらない汎用的な座標処理で，画面表示に必要なデータのみを差分転送できるようになり，表示内容によらずにフレームワークを適用可能とした。なお，画像の差分取得に対応するため，サーバサイドで描画した画像は任意の大きさのブロックに分割して管理する。クライアントへの画像の転送はこのブロック単位で行う。

3.2 複数デバイス対応のためのイベントハンドリング共通化

提案するコンポーネントは各種モバイル端末とブラウザをサポートする。しかし，UI 操作に対するイベントはプラットフォームごとに異なるため，全てのデバイスおよびブラウザの組み合わせを網羅しようとするとそれぞれのイベントハンドラの実装が必要となり開発工数がかかる。そこで，図 4 のようにコンポーネント内部でブラウザから発行されるイベントを抽象化し，デバイスによらない同一のイベントインタフェースをアプリケーション開発者に提供する仕組みとした。

UI 応答として，スクロール，ズーム，ドリルダウン(画像の入替)をサポートする。これらの操作はユーザのマウスホイール操作，スワイプ操作およびピンチ操作に対して任意の組み合わせで紐付け可能とした。つまり，マウスホイール操作に対してスクロールを紐付けると，ユーザのマウスホイール操作に応じてコンポーネントが表示している画面をスクロールする処理を行う。同様にピンチ操作に対してズームを紐付けるとユーザのピンチ操作に応じてコンポーネントが画像の拡大縮小処理を行い，ユーザの画面操作に対する応答を実現する。これにより，アプリケーションの開発者はこれらの汎用的な UI 操作に対する応答をコーディングレスで実現できる。UI 操作がされた場合はフレームワークがブラウザ上の DOM を操作することで，画像の表示位置などを変更し操作への応答を実現する。また，マウスの右クリックやロングタップなどの UI 操作に対してコンテキストメニューの表示を紐付けることができるようにし，開発者はコンテキストメニューの表示や消去などの制御に注力しなくともよい構造とした。

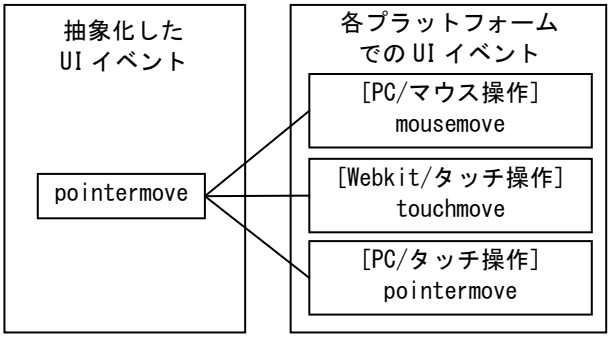

図 4 イベントの抽象化

3.3 コンポーネントフレームワークとしての実現

提案方式を様々なコンポーネント開発に適用可能とするためにコンポーネントフレームワークとした。その構成を図 5 に示す。

提案するコンポーネントフレームワークはサーバおよびクライアントにまたがった形態をとり，サーバ上で描画した画像を，データ通信部を介してクライアントに送信する。クライアントは画像を受信すると画面表示処理部が Web ブラウザ上に指定された領域内に画像を表示する。また，初回表示までの時間を短縮するためにクライアントの画面表示に必要な領域だけを先に転送し，残りの画像はクライアントの画面スクロールなどにあわせて差分取得する制御を行う。

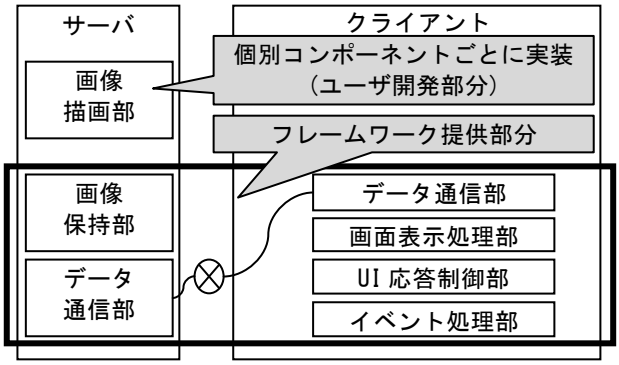

図 5 コンポーネントフレームワークの構成

図 5 でフレームワークが提供する部分は太枠内の部分である。枠外のサーバ上での画像描画部および描画に必要

なデータの取得および加工などの処理は開発するコンポーネントごとにフレームワークのユーザ(開発者)が実装を行う。つまり，提案するフレームワークを利用することで，開発者はサーバ上で画像を描画するのみで，PC や各種スマートデバイスの画面上にその結果を表示することができるようになる。

提案するコンポーネントフレームワークを利用することで，従来は処理負荷が高くスマートデバイスで表示が困難であった課題を持つコンポーネントが Web ページ上で表示可能となる。また，コンポーネントフレームワークとしてライブラリ化することにより再利用可能となり類似の課題を持つコンポーネントの開発工数低減に貢献できる。なお，今回の試作にあたり，サーバサイドは.NET Framework(C#)のアセンブリ，クライアントは JavaScript ライブラリとして試作した。フレームワークを Web ページ内に組み込み利用する方法は付録に記載する。以下，コンポーネント内で実行される処理に関する説明を行う。

3.3.1 画像保持部

Web システムで用いられる HTTP[3]プロトコルはステートレスなプロトコルであるため，画像データの差分送信を実現するためには，その永続化が必要となる。そこで，ステートフルなプロトコルである WebSocket[4]を用い，サーバのメモリ上に画像を保持することとした。これにより，差分取得の際にも初回接続時に生成した画像を参照できるため，差分取得時の画像再生成が不要となる。

3.3.2 データ通信部

サーバとクライアント間に WebSocket による接続を生成し，画像データ等の送受信を JSON[5]形式で行う。通信部分はアプリケーションの開発難易度低減および品質確保を目的とし，内部に隠蔽した。これにより，アプリケーションの開発者は WebSocket の通信制御に関する処理を考える必要がなくなり，コンポーネント導入に対する障壁が小さくなる。ただし，ネイティブアプリケーションなどの開発を可能とするため，通信プロトコルの仕様は公開とした。

3.3.3 画面表示処理部

JavaScript を用いて動的に DOM を操作し画像を HTML 文書内に挿入することで，サーバから受信した画像を Web ブラウザ上に表示する。画像の表示位置はスクロール位置やズーム倍率などにあわせて計算する。また，クライアントの画面上にダイアログやコンテキストメニューを表示する機能を提供する。なお，メニュー関係は jQuery UI[6]を用いて実現した。

4. コンポーネントの評価

本章では，提案したコンポーネントフレームワークを実際のアプリケーションの Web 化に適用した想定で，機能面および性能面の評価を行った。商品の在庫状況を散布図やグラフ表示などで可視化するコンポーネントを題材とし，スマートデバイス上からビューアとして利用可能な Web アプリケーションを作成する想定で評価を行った。コンポーネントフレームワークはマルチデバイス，マルチプラットフォーム対応のため，適用にあたり課題(1)で挙げた問題は発生しない。

4.1 試作アプリケーション概要

本コンポーネントフレームワークの評価にあたり，スマートデバイス上でデータを閲覧するビューアアプリケーションを試作した。これは図 6 に示す散布図コンポーネント(散布図画面)，多数のグラフを同時に表示するモニタコンポーネント(一覧表示画面)，グラフコンポーネント(グラフ画面)の 3 種類のコンポーネントを含む画面を持つ。散布図コンポーネントでは在庫がどのような特徴を持っているかを可視化するために 2 軸のグラフ上に点をプロットする。一覧表示コンポーネントでは品目ごとの在庫推移状況を一覧表示し，問題となる在庫を可視化する。また，グラフコンポーネントでは品目ごとの詳細な在庫状況を表示する。

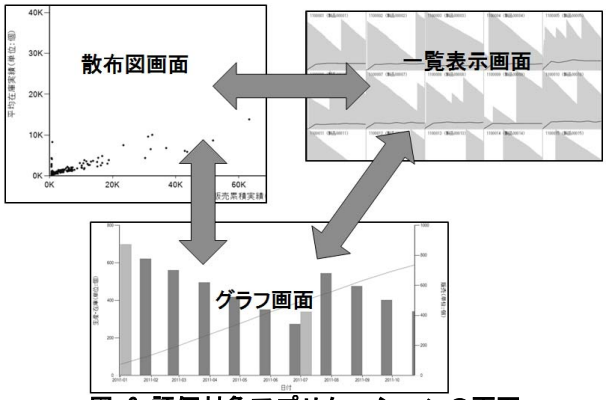

図 6 評価対象アプリケーションの画面

クライアントが接続すると，はじめに散布図コンポーネントを持つ画面が表示され，プロットされた点の選択やコンテキストメニューの利用により他のコンポーネントを持つ画面の間の遷移を行う。画面遷移時にはコンポーネントのドリルダウン機能を利用して，サーバ上で新規に画像を生成し，クライアントの画面を更新する。なお，サーバ上ではクライアントに表示する画面全体を一括生成する。

4.2 コンポーネントフレームワークの機能評価

4.2.1 機能評価条件の設定

コンポーネントフレームワークを用いて既存のスタンドアロンアプリケーションを Web 化するにあたり，フレームワークが十分な機能を提供しているかを評価した。

表 3 各画面での機能要件

画面	散布図	一覧表示	グラフ
固定領域	左，下	なし	左，下，右
画面の原点	左下	左上	左下
ホイール	ズーム	スクロール	ズーム
スワイプ	スクロール	スクロール	スクロール
ピンチ	ズーム	ズーム	ズーム
データの選択	単一選択：クリック/タップ 複数選択：範囲をドラッグ 選択時は該当するデータをハイライト表示する		なし
画面遷移	コンテキストメニュー		
	データのダブルクリック		ズーム

評価対象とするコンポーネントの機能要件を表 3 に示す。各画面は基本的にビューアとしての機能を提供し，画面内に表示されているデータの選択やコンテキストメニューを用いて画面間を遷移する設計とした。

4.2.2 機能評価結果

コンポーネントの機能要件に関連するフレームワークの機能は表 4 の通りである。これをもとに表 3 に挙げた機能要件をフレームワークが提供する機能を用いて実現可能か否かをまとめた。その結果を表 5 に示す。

表 4　フレームワーク提供機能

機能	説明
固定領域	画面の上下左右に対して固定領域を設定可能。また，固定領域とする位置の組み合わせは任意に指定可能。
画面の原点	画面を表示する際にスクロールなどの原点とする位置は左上，右上，左下，右下の 4 種類から選択可能。
UI 操作に対する応答	マウスホイール，スワイプ，ピンチ操作に対してスクロール，ズーム，ドリルダウンを任意に紐付け可能。
イベント処理	クリック，タップ，ドラッグなど基本的な操作に対して抽象化したイベントインタフェースを提供する。
メニュー表示	ダイアログおよびコンテキストメニューを表示可能。
画面遷移	コンテキストメニューやクリックなどのイベント内で画面遷移が可能。

今回提案するコンポーネントフレームワークを用いることで，画面遷移をもつビューアを想定したコンポーネントの機能要件を実現できることが分かった。また，実現方法についても，フレームワークの機能として提供しているものを基本的にそのまま利用するだけでよく，複雑な実装などは不要であることが分かった。しかし，ズームアウト時の画面遷移が現状のままでは実現困難なことが分かった。これはスクロールやズームなどの画面更新を伴う処理の中での画面遷移はフレームワークの設計時に想定していなかったためである。今回の評価結果をもとにフレームワークの改良を検討していくこととする。

表 5　機能要件の実現可否

要件		実現可否
固定領域の位置	左，下	○
	なし	○
	左，下，右	○
画面の原点	左下	○
	左上	○
UI 操作	ホイールでのズーム	○
	ホイールでのスクロール	○
	スワイプでのスクロール	○
	ピンチでのズーム	○
選択操作	クリック/タップ	○
	ドラッグ	○
	対象のハイライト表示	○
画面遷移	コンテキストメニュー	○
	ダブルクリック	○
	ズーム	△

4.3 コンポーネントフレームワークの性能評価

4.3.1 性能評価条件の設定

性能評価では図 6 に示す散布図画面と一覧表示画面を対象とした。これらの 2 つの画面は表 6 に示す特徴をもつ。散布図画面では 1 枚の散布図上に各データに応じた点をプロットするため，サーバ上で描画する画面の解像度はデータ数によらず一定となる。しかし，散布図の描画時にはデータ数に応じて点の描画処理を実行するため，描画に必要な時間はデータ数に応じて増加する。一方で，一覧表示画面は，データごとにグラフを 1 枚描画し，そのグラフを全て並べて一覧表示する画面構成となるため，グラフ 1 枚を描画するための時間はほぼ一定となる。しかし，データごとに 1 枚のグラフ画像を生成しそれを一覧表示するため，サーバ上で生成する画像全体の解像度はデータ数に応じて増加していく特徴をもつ。性能測定では，表 6 に示す異なった特徴を持つモデルに対し，コンポーネントフレームワークを適用した際の性能を測定した。

表 6　データ数に対する各画面の特徴

	散布図画面	一覧表示画面
画面全体の解像度	一定	データ(グラフ)数に応じて増加
グラフ 1 枚あたりの描画時間	データ(プロット)数に応じて増加	ほぼ一定

今回の性能評価では，はじめに散布図画面を表示し，その後に在庫一覧を表示するために一覧表示画面に遷移したというユースケースを想定して上記 2 種類の画面を表示する際にかかる時間を測定した。つまり，クライアントの初期接続時に散布図画面を表示するまでの時間と，その状態からコンポーネントのドリルダウン機能を用いて一覧表示画面へと遷移し在庫状況を一覧表示する場合の 2 つについて，入力データ数を変化させ計測した。なお，サーバ上でのデータの読み込みは完了しているものとする。処理時間はサーバおよびクライアントについてそれぞれの処理ごとに集計し，フレームワークの処理にかかる時間と各コンポーネント固有の処理にかかる時間を分離可能とした。具体的には表 7 に示す処理項目に分けてそれぞれの処理時間を計測した。

表 7　画面表示における処理項目一覧

処理項目	説明	担当
画像生成	サーバ上で画面を描画し画像を生成する処理	C
画像処理	散布図画面： 生成した画像をブロック状に分割し配列に格納する 一覧表示画面： 品目ごとに生成したグラフを配列に格納する	C
Server 処理	接続要求の処理や JSON 形式で応答を返すなどサーバ側の処理	F
NW 通信	通信時間	F
Client 処理	画像を受信して DOM に挿入するなどクライアント側の処理	F

※C: 個別コンポーネント側，F:フレームワーク側を示す

クライアントからの接続要求があるとフレームワークが各コンポーネントに通知を行う。それを受けた各コンポーネントが固有の描画処理を行い，画像を生成する。さらに，データの差分取得に対応するため，画像をブロック状に分割した後に，配列に格納してフレームワークへと渡す。フレームワークはブロック状に分割された画像を受け取った後，クライアントの要求に応じて画像ブロックを送信する処理を行う。クライアントは受信した画像ブロックを DOM に挿入することで，Web ページ上に表示する。今回の性能評価では，サーバで生成した画像が表示されるまでの時間を計測対象としているため，上記処理のうちサーバサイドでの画像生成および画像分割が個別コンポーネント側の処理時間となる。

表 8　性能測定環境

サーバ	Windows Server 2012 R2 Intel core i7 4770 3.4GHz / 4GB
クライアント	Android 4.4.4 APQ8064 QuadCore 1.5GHz / 2GB Firefox 表示領域の解像度　970×423 ピクセル
ネットワーク	Wi-Fi 接続(IEEE802.11n)

性能測定に利用した環境の詳細を表 8 に示す。入力とするデータ数は 500,1000,2000,3000,4000,5000 品目とし，それぞれ 12 カ月分の在庫推移情報を持つものを用いた。また，測定は 10 回行い，それらの平均値と標準偏差を求めた。散布図画面の表示では目盛を表示するための領域として，画面の左側と下側に固定領域を設定し，それぞれの大きさは画像全体の 10%および 15%分とした。また，クライアントの画面解像度と同じ解像度を持つ画像をサーバ上で生成し，その画像を 100×100 ピクセルのブロック状に分割した。一覧表示画面では，品目ごとに 240×240 ピクセルのグラフを生成し，それを 1 つの画像ブロックとした。画像はフレームワークに渡すために配列に格納し，配列オブジェクトとして渡した。

4.3.2　性能評価結果(散布図画面)

サーバ上で生成する画像がデータ数によらず，常に同一解像度となるモデルである散布図画面での性能測定結果を図 7 および表 9 に示す。

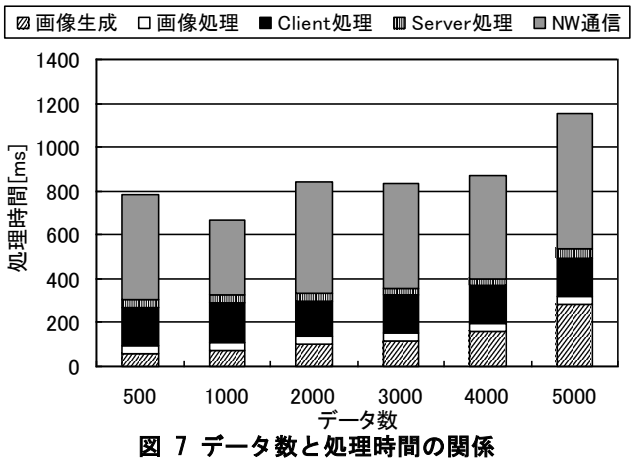

図 7 データ数と処理時間の関係

表 9　処理時間一覧[ms]

データ数	画像生成	画像処理	Client	Server	通信
500	60 (16)	33 (5)	179 (8)	31 (8)	480 (123)
1000	72 (11)	36 (1)	182 (16)	34 (4)	345 (51)
2000	98 (12)	40 (5)	161 (12)	34 (3)	512 (121)
3000	118 (17)	34 (8)	176 (14)	31 (11)	475 (86)
4000	157 (24)	36 (3)	176 (9)	30 (4)	468 (62)
5000	285 (188)	37 (1)	170 (14)	47 (17)	616 (200)

下段括弧内：標準偏差

処理時間を見ると，処理性能が低いスマートデバイスの場合でも実用的な 1 秒程度で画面表示が完了することが分かる。また，データ転送量も PNG 画像 1 枚で画面表示が可能となるため，サーバ上に保存されているデータサイズに依存することなく，Web ページの閲覧時と同等のデータ転送量のみで画面を表示することができる。以上より課題(2)および(3)が解決できている。また，このモデルの場合，グラフ 1 枚が出力となるため課題(4)のデータの分別については評価の対象とはならない。

次にデータ数の変化による処理内容ごとの処理時間の傾向を表 10 に示す。

表 10　処理時間の傾向

処理時間の傾向	処理内容	処理担当
増加する	画像生成	個別コンポーネント
変化しない	画像分割	個別コンポーネント
	Client 処理	フレームワーク
	Server 処理	フレームワーク
	通信	フレームワーク

データ数の増加に伴い全体の処理時間は増加傾向にあるが，その増加要因は画像生成処理が占めており，他の部分はデータ数の増加によらず処理時間がほぼ一定となった。画像生成処理では，入力されたデータを散布図画面内のしかるべき位置に描画する処理が行われるため，描画すべきデータ数の増加に伴い，処理時間も増加したものと考えられる。それ以外の処理については，画像生成後に行われる処理であるため，処理対象となる画像の解像度が常に一定となる今回のようなモデルの場合では，各々の処理時間も変化しないことが確認できた。

また，処理種別に着目すると，フレームワーク側の処理はデータ数によらず処理時間がほぼ変化しないという結果が得られた。今回の条件では，フレームワーク側で処理すべき画像の解像度が入力されたデータ数によらずに同じになるため，処理負荷が変わらないためである。今回のようなモデルでは，フレームワーク側の処理の中で大幅な性能劣化を引き起こすことはないため，全体の処理時間を短縮するためには，個別のコンポーネント側の処理改善に注力すればよいことが示せた。

4.3.3 性能評価結果(一覧表示画面)

次に，サーバ上で生成する画像がデータ数に依存し変化するモデルである一覧表示画面での性能測定結果を示す。

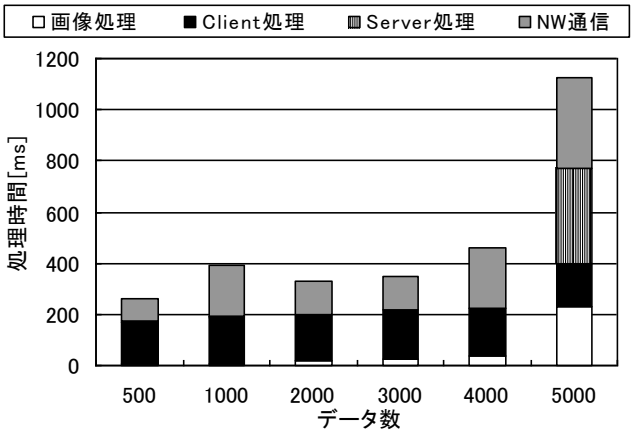

図 8 データ数と処理時間の関係

表 11　処理時間一覧[ms]

データ数	画像生成	画像処理	Client	Server	通信
500	2594	4	166	1	89
	(40)	(0)	(37)	(0)	(60)
1000	5155	9	180	1	202
	(87)	(1)	(20)	(0)	(202)
2000	10386	16	183	2	129
	(248)	(6)	(15)	(1)	(71)
3000	15825	26	191	3	127
	(161)	(2)	(17)	(1)	(85)
4000	21873	38	184	3	233
	(328)	(1)	(17)	(0)	(192)
5000	32696	229	168	372	358
	(7568)	(194)	(9)	(639)	(394)

下段括弧内：標準偏差

散布図画面を表示した後にフレームワークのドリルダウン機能を利用して全品目の在庫状況一覧画面を表示するまでの時間を計測した。その結果を図 8 および表 11 に示す。なお，一覧表示画面においては，品目ごとにグラフを 1 枚描画するためサーバ上での画像生成時間が処理時間の大部分を占めるようになった。そのため，図 8 には画像生成以外の部分を抜き出したものを示す。

処理時間を見ると，描画するグラフの数に応じて処理時間が延びていることが分かる。描画するグラフが 500 個の場合は約 3 秒程度で画面が表示されるが，グラフが 5000 個になると，画面表示までに 30 秒程度待たされることが分かった。画像生成処理が全体の処理時間の大部分を占める傾向となっており，画面表示までの時間を短縮するためには，画像生成の絶対時間を短縮する必要がある。以上より，課題(2)の応答性能については，初期表示時における描画処理の並列化などの工夫が必要となることが分かった。ただし，グラフ画像は全て描画が完了しているため，クライアントのスクロール操作の際に Canvas を使用していた場合に必要であった再描画処理が不要となり，十分な画面応答性能を得ることができた。

測定結果の標準偏差を見ると，描画するグラフが 5000 個となった場合に大きな値となることが分かった。これは，グラフの描画に必要なメモリ量の増加に伴い GC (Garbage Collection)が発生するようになり，実行時のメモリの状況によってその処理時間がばらつくようになるためと考えられる。次に，処理内容ごとの処理時間の傾向を表 12 に示す。

表 12　処理時間の傾向

処理時間の傾向	処理内容	処理担当
増加する	画像生成	個別コンポーネント
	画像分割	個別コンポーネント
変化しない	Client 処理	フレームワーク
データ数 5000 の場合に増加する	Server 処理	フレームワーク
	通信	フレームワーク

処理ごとの時間を比較すると，全品目分のデータをグラフで一覧表示する場合，画像生成および画像処理に関わる時間がグラフの数に応じて増加する傾向があることが分かる。これは，グラフ画像の生成回数および画像を配列に格納する回数が，その数の増加にあわせて増えるためである。一方で，グラフの数が 4000 程度までは，クライアントおよびサーバでのフレームワークの内部処理時間はサーバ上で生成する画像の解像度によらずほぼ一定となった。その理由を表 13 にまとめた。フレームワークでは，サーバで個別コンポーネントが生成した画像を受け取った後は，クライアントの画面表示に必要な部分だけを送信し，Web ブラウザ上に表示する構造のため，サーバ上で生成した画像全体の解像度に依存せずに，ほぼ一定の負荷で画面表示を行うことができるためである。これにより課題(3)のデータ転送に伴う待ち時間の問題が解決できたといえる。また，課題(4)に挙げたデータの分別については，データを画像化することで，画面表示に必要となる座標情報をクライアントがサーバに通知し，サーバが該当する画像をクライアントに返す処理として容易に実現できている。

表 13　処理時間が変化しない理由

処理内容	理由
Client 処理	クライアントサイドでは画面に表示する分の画像データに関する処理だけを行うため，実行速度はサーバ上の画像解像度に依存しない
Server 処理	画像本体に対する処理は行わず，クライアントからのリクエストに応じて配列として保持している画像の中から 1 枚を返すだけの処理のため大きく性能が劣化することがない
通信	クライアントには画面内に表示するデータだけを送信するため，サーバ上にある画像全体の解像度に依存しない

また，グラフの数が 5000 個となった条件下で大幅な性能劣化が発生した。この場合は，サーバ上で生成される画像の解像度が 30 万×970 ピクセルとなり，1.16GB 程度のメモリを消費したためサーバの物理メモリが枯渇し，大規模な GC が発生したと考えられる。画像の生成，分割処理で GC が発生した場合，巨大な領域の解放処理やメモリの断片化解消処理などが実行されるようになり，サーバ上のスレッドが長時間ブロックされ，応答性能が大幅に劣化したと考えられる。フレームワークの適用時には

メモリ使用量を見積もり，負荷に対応できる数のサーバを用意するなどの対応が必要である。また，現在はサーバ上で画面全体を一括生成しているが，表示に必要な領域のみを先に生成し，残りを遅延生成することで画面表示までの時間を短縮するなど，フレームワークの改良も進める必要がある。

5. おわりに

描画に多くの計算量が必要となる画面を，計算リソースに制限があるスマートデバイス上で描画した場合，クライアントに多くの負荷がかかり十分な UI 応答性能が得られない。そこで，描画処理をサーバ上で実行することでクライアントの負荷を下げる方式をとるコンポーネントフレームワークを提案した。サーバで画像化することで画面表示に必要なデータを座標情報のみで容易に分別可能となり表示までの待ち時間を短縮できる。フレームワークの評価を目的として，散布図やグラフの一覧表示画面で在庫状況を可視化するアプリケーションの Web 化に適用し，フレームワークの機能面に大きな問題がないことを示した。また，データ規模を変えた性能評価を行い，サーバサイドで生成する画像の解像度によらずに必要な部分のみを差分転送するため，GC が発生しない条件の下で，ほぼ一定した性能を維持できることを示した。

提案したコンポーネントフレームワークはマルチプラットフォーム，マルチデバイス対応であるため，Web ページ内に容易に組み込むことができる。これにより，複雑な UI をコンポーネント化することで画面サイズなどのデバイスの特性に応じた Web アプリケーションのデザインが可能となる。また，デバイスによらない単一のプログラミングインタフェースを提供することで，アプリケーションの開発時にデバイスの差異を考慮する工数が不要となり，ソースコードの共通化による保守性の向上に貢献することができる。さらに，外出時の不安定な接続環境においても Web システムの利用が可能となり業務の効率化が期待できる。

参考文献

[1]HTML5 キャンバスのレイヤー化によるレンダリングの最適化, http://www.ibm.com/developerworks/jp/web/library/wa-canvashtml5layering/, Accessed 2015/6.
[2]内海 宏律, 菊地 大介, 浦邊 信太郎, 齋藤 邦夫, 手塚 大, “サーバサイドイメージレンダリングによるウェブ UI コンポーネント・フレームワーク”, 情報処理学会 第 77 回全国大会論文集分冊 3, pp.65-66, (2015).
[3] Hypertext Transfer Protocol, http://tools.ietf.org/html/rfc1945, Accessed 2015/3.
[4]The WebSocket Protocol | IETF, https://tools.ietf.org/html/rfc6455, Accessed 2015/3.
[5]The JavaScript Object Notation (JSON) Data Interchange Format, https://tools.ietf.org/html/rfc7159, Accessed 2015/3.
[6]jQuery UI, https://jqueryui.com, Accessed 2015/3.
[7]jQuery, https://jquery.com, Accessed 2015/3.

付録

提案するコンポーネントフレームワークのクライアントサイドは JavaScript ライブラリとして提供する。つまり，jQuery[7]のように広く普及している JavaScript ライブラリ同様に Web ページに組み込んで利用する形態となる。

```
【HTML】
<div id="ganttdiv" style=
"width:1000px;height:480px"></div>

【JavaScript】
var gantt = new JP.HSE.CUIC.cuicClient(
    document.getElementById("ganttdiv"),
    {
      autoConnect: true,
      url: “ws://・・・・・・・・・",
      (以下，略)
    });
```

図 9 フレームワーク適用ソースコード例

コンポーネントフレームワークを利用する際は，図 9 のように Web ページの任意の領域に div 要素を設定するとその中にサーバで描画した画像が表示される。Web ページ内に容易に組み込み可能な UI コンポーネントであるため，図 10 に示すように Web ページの任意の領域に対して適用することができる。これにより，従来技術では表示が困難であった複雑な画面をスマートデバイス上で表示可能とした。

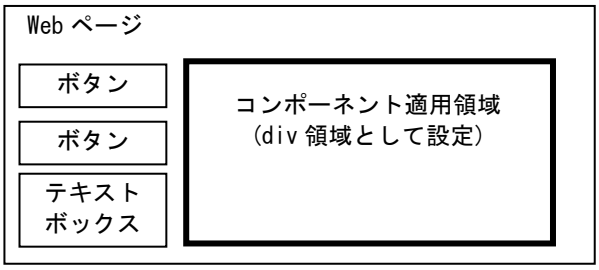

図 10 コンポーネント適用による画面構築

フレームワークを利用したコンポーネントを適用した画面では，図 10 のように，Web ページ内に設けた任意の div 領域内にサーバで生成した画像を埋め込む形となるため，周囲のボタンやメニューなどはデバイスに応じたものを利用できる。例えば，PC 向けにはタブやメニューを持つ画面を提供し，スマートデバイス向けには小さい画面でも操作しやすいリスト形式のメニューなどを提供することができる。これにより，スマートデバイスから PC 向けのアプリケーションをリモート接続で利用した場合のように，操作のしにくい UI ではなく，デバイスにあわせた UI を提供できる利点がある。また，コンポーネントを適用した領域内では，デバイスに応じた操作体系を提供する。たとえば，PC で Web ページを閲覧した場合はマウス操作に応じてスクロール処理などを行い，タッチパネルを搭載した端末に対してはタッチ操作に対してスクロールやズームなど，適切な画面応答を行う機構をフレームワークが提供する。