長野大学紀要　第31巻第３号　１—14頁（281—294頁）2010

教室のためのソフトウェア技術（Ⅰ）
—Webと Ajaxの可能性と限界—

Software Technologies for Classrooms（Ⅰ）
—On the Feasibility of the Web and Ajax—

平　岡　信　之*
Nobuyuki Hiraoka

1　はじめに

筆者は大学でコンピュータにかかわる教育に携わるようになり、教壇に立つ業務に伴って、教室における情報環境について考えることも日常業務の１つになっている。この領域においては、たまたま約15年前に大学の情報実習教室の情報装備の概念設計にかかわる機会を得たこともあり、その時に調査したり考察したりしたことが筆者の思考の土台になっている[1]。しかしその後、情報教育をとりまく外部環境は大きく変化した。当時の情報実習教室はデスクトップパソコンが主役であったのに対し、現在では、ノートやサブノート等のポータブルな機器が主役となり、多くの大学でノートPCを個人で持ち歩くことが主流になっている。教室のデスクトップパソコン、すなわち、大学で所有し、ソフトウェアの選択や設定についての権限も大学で持っている、仕様の統一されたコンピュータ、から、ソフトウェアの選択や設定についての権限を個人で持ち、（機器の統一が完全に行われたとしても）入学年度によりメーカーや仕様（OSの版も含めて）が変化する、不統一な機器群、へと変化したことになる。教育環境であるPCの中は、いわば教える側からも不可触な領域になった。

こうした状況に対応しつつ、せっかく各学生に携帯してもらっているPCを活用して教育効果を高めるには、どんな環境整備を行えばいいか、これを課題として、ソフトウェアとネットワーク技術を活用した取り組みを始めた。この取り組みは、以下の２つのテーマに沿って進めている。

a　多様なパソコンにどう対応するか＝＞仮想環境の活用

b　机上にある各学生のパソコン画面をどう活用するか＝＞ネットワークを通じたオンラインプレゼンテーションの発展形の模索

研究は進行中であるが、その過程で得た成果や知見をこの誌面をお借りしてシリーズにして報告していきたい。第一回の本稿では、上記のbの機能を、Web技術に基づいて実現するための、Ajax技術についての調査を行った報告を行う。

2　想定するアプリケーション

2.1　教室の現状と課題（研究の狙い）

教室には伝統的に黒板（または白板）が装備されており、従来の教室スタイルでは、話し言葉による情報伝達を主軸にして、黒板を使って視覚的な補足を行っている。もう少し細かく分析すると、以下のような情報伝達チャンネルが使われていることになる。

a　音声による伝達＋顔や身振りなどの補助的情報

*企業情報学部准教授

b　黒板による文字や図を用いた視覚情報
c　配布物による視覚情報

以上が下り方向の情報伝達だとすると、逆に、上り方向には、

d　声、視線、指示に従う動作、といった意識的でないリアクション
e　提出物、メールなどによる意識的リアクション

といった形で、学生から先生へのフィードバックが返されていると考えられる。

上記のうち、aやbは記録として残せない（インテリジェント白板やビデオ記録といった装備のある環境なら可能だが、そういう装備を持つ教室はまだまだ少ない）し、cは、記録としては優れていても、話の流れに沿って受講者の着目する視線を話者側からコントロールすることができない。昨今の教室でのPC利用の主流はパワーポイントに代表されるプレゼンテーションだが、このプレゼンテーションは、予定の筋書きにそった情報提示には便利なツールであるが、その反面、質問への対応などアドホックな情報提示の際には、情報生成のための労力がまだまだ大きく、そういう場面ではbの黒板に頼ることが多くなる。といった具合に、様々なチャンネルを併用する結果として、受講者の視線がスクリーン、黒板、配布物、自分のPCなど、数か所の間で行き来することになる。その結果として、授業では「次にｘｘｘを見てください」といった視線コントロールのための、いわば内容から離れた発言が増え、授業の効率を低下させる一因となっていると考えられる。

こうした問題を緩和するための技術的アプローチとして、上記b、cの情報を学生パソコンの画面に集約するべく、オンラインプレゼンテーションのためのソフトを開発したいと考えている。そのソフトで、dやeの上り方向の情報の一部も伝達できることを狙いとしたい。

2.2　機能の概要

ノートPCとLAN以外に特殊な装備を持たない教室環境で、学生のPC画面に先生機にある情報を映し出す方法としては、VNC[1]をはじめとする遠隔操作ソフトを表示専用モードで使うというやり方がある。このソフトは逆方向で使用すると学生機の画面を先生が監視したり必要に応じて介入や操作補助を行うこともでき、教室で効果を発揮するソフトである。しかしこのソフトでは、前述のようにリアクションの検出を行うことができない（例えばVNCで教卓機を表示専用でないモードで共有許可すると大変な混乱になることが想像できるだろう）。学生のリアクションを検出するためには、学生機画面に、文字や図といった情報提示だけでなく、インタラクション部品もまた画面に表示し、その部品を通じた応答を先生機に返送させるようにしたい。それを含めて、このソフトに盛り込むべき機能としては以下のものを想定することとする。

1　文字や図を適切な場所に表示する
2　インタラクション部品を表示し、応答を返送する
3　手描き線によるアノテーションをそれら上に重ねて表示する
4　ユーザの無意識な動作を情報として返送する
5　それぞれの文字や図やインタラクション部品といった表示要素の表示開始や状態変更は、
　・先生の操作したタイミング
　・各学生がインタラクション部品に操作を行ったタイミング
　のいずれかで、個別に設定できる

2.3　技術の選定

上記の機能を持つアプリケーションを作成するにあたっては、大別すると2つのアプローチが考えられる。

a　独立したアプリケーション。通信にはTCPソケットを使用し（トランスポート層）、プロトコルは独自設計し、GUIは適当なツールキットを使う。
b　Webアプリケーションとして作る。この場合、通信はWebと同じHTTPを主に使用し（アプリケーション層）、GUIはWebに依存する。

先に述べた、受講者の携行するPCの多様性を考慮すると、bによるアプローチのほうが高いイン

ターオペラビリティを担保することができ、メリットが大きいと考えられる。ただし、Webベースのシステムを作る際には、Webアプリケーションであるが故の様々な制約を知った上で、その制約の範囲での開発を行う必要がある。そこで、Webアプリケーション開発を前提としておいた上で、しかし開発にかかる前に、前節の要求仕様に基づいて、その実現可能性についての技術評価を行うこととした。

3　Web技術の概要

3.1　ブラウザの動作（の中心部）

ブラウザはHTMLで書かれたドキュメントを読みこみ、その内容に基づいて内部にデータ構造を生成し、それをCSSに基づいたルールに従って画面に描画する。その内部データ構造はDOMとして認識されている。現在のJavaScriptにはそのDOMやCSSを操作する機能が装備されており、ドキュメントの構造や表示の仕方をダイナミックに変化させることができる。JavaScriptは（パソコンから見て）外部から降ってくるファイルに含まれるものであり、悪意のあるコードがパソコンに害を与えることのないよう、JavaScriptが操作することのできるオブジェクトは、当該ページの文書にかかわるものに限定されている。

3.2　Ajax[1]とは

Web2.0という用語が有名になったが、そのWeb2.0の中核となる技術の1つがAjaxである。GoogleMapがその代表例として知られている。Ajaxでは非同期通信を用いて、ページの遷移を伴わない（サーバとの間の）通信を行って、1つのページを表示しながら、それと並行して、様々な処理を行うことが可能になっている。Ajaxの導入により、以前のWebの、紙芝居的にページがどんどん切り替わる画面のイメージが払拭され、使い勝手のいいWebアプリケーションが多数作られるようになった（使い勝手の悪いものも多数作られているが）。

3.3　サーバプッシュ技術

たまたま本研究では、とりあえず教室という閉じた環境、いわゆるイントラネットを想定していて、その環境下では通信プロトコルの制約は少なく、自前の通信方式によるアプリケーションの開発も無理ではない状況なのだが、インターネットで広域のサービスを展開する立場からすると、プロトコルをHTTPに限定したりクライアントソフトウェアをWebブラウザに限定したりする動機はより強いものになることは容易に想像できる。様々なサービスをWebアプリケーションに集約していこうとする動きの中で、サーバからクライアントに向けて連続的にデータを送ったり（ストリーミング）、サーバのタイミングでデータを送出したり（サーバプッシュ）する技術のニーズは高いようである。そのための汎用技術として、CometやAPEといった技術が提唱され、また活用もされてきている。そういったパッケージされた技術は、いわゆるC10K問題[ii]（クライアント1万台問題）への対応も視野に入れ、性能が重視される場面を想定して作られているものが多いが、その反面、システムは複雑になり、サーバに依存する技術選択を行ってしまうことになる。本研究で想定される教室環境は、C10K問題の危険にさらされる広域のインターネットアプリケーションではなくクライアント数がせいぜい数十台規模のアプリケーションなので、こういった技術の採用は当面は見送ることとしている。

3.4　その他の技術

a　アプリケーションフレームワーク

Webアプリケーションの開発を省力化したり、サーバ側のプログラムとクライアント側のプログラムを統合した形で開発できるようなアプリケーションフレームワークが多数作られ使われている。楽ができるなら勿論採用したいので、これから鋭意調査の対象として行きたいが、一般にはフレームワークはパターン化したページの作成を得意分野としているようであり、本研究の開発に適したものが見つかるかどうかは不明である。

b　RIA（Rich Internet Applecation）

Ajaxによるアプリケーションもこれに含まれる場合もあるが、一般には、javaアプレット、Flash、Flex、Silverlightなど、ブラウザ上でプラグインとして動作するプログラムで表示・再生するような動的コンテンツを用いたWebアプリ

ケーションを言う。それぞれ、記述力が高く、それこそ何でもできそうな気のする魅力を持った環境である。ただ、これらのRIA勢力が現在は統一なくしのぎを削っている状態であるため、現時点で1つの勢力の傘下に入った形での開発を行いたくないため、あえてRIAの存在に目をつぶっているところである。

4 調査の方法と環境

4.1 調査の方針

前章に述べたアプリケーションをWebで実現する際に、実現可能性が未確認であった要素技術を選び出し、それぞれについて、実現の方法をネットで検索可能な情報源をあたることで調査し、複数の候補がある時にはその比較選定を行った上で、評価用の簡単なプログラムを作成し動作確認を行うとともに問題点の抽出を行うこととした。その要素技術については次章で報告する。

4.2 道具立て

a 開発言語と開発支援ライブラリ

ブラウザで動作できるスクリプト言語は多数あるものの、標準的に使用される言語はJavaScript[iii]であり、<script>タグのデフォルトの言語にもなっているため、これを採用する[2)]。JavaScriptでAjax開発を支援する様々なライブラリが流通しており[iv]、このうちのいずれかを（或いは複数のものを）使用することで開発効率は上がると思われる。しかし、これらのライブラリは突き詰めればJavaScriptネイティブの機能の呼び出しに帰着するものであり、本調査では、ライブラリに依存する開発として出発することにならないよう、極力JavaScirptネイティブの技術に限定しての調査を行うこととした。ただし、Prototype.jsについては、すでに若干の使用経験があり素性がわかっていることと、機能はそう高くないものの、必要最低限の支援を行う標準的なライブラリとして認知されていることから、このライブラリに限っては必要に応じて使用することとした。

一方の、サーバ側で動作するプログラムについては、ruby言語[v]を用いることにしている。オブジェクト指向で、記述力が高く、支援ライブラリが充実していることなど、rubyの利点は様々あるが、第一には、筆者が使い慣れていることがポイントであった。支援ライブラリ（組込でないクラス）としては、rubyでCGIを作ったり分散オブジェクトを使ったりする場面で、drubyとerubyを用いた（クラス名としては、それぞれDRBとERB）。

b 対象とするブラウザ

本研究の志す用途を考えるとクロスブラウザな（ブラウザの種類を問わない）技術開発を行うのが理想ではあるが、当面はクロスブラウザであることは第一条件とはせず、1つのブラウザを基準に調査を行うこととした。対象としては、以下の理由により、firefoxの最新版を選び、必要に応じて他のブラウザでも動作確認を行うようにした。

・firebugs（後述）がfirefoxに対応しており、また、後述のfiddlerもfirefoxの画面上で機能のオンオフが切り替えられるなど、開発者向けブラウザとして優れている。

・履歴管理など多数のプラグインがあり使いやすいため筆者自身が日頃多用している。

・事実上の標準OSであるWindowsに添付されており、その結果として最もシェアの高いブラウザとして、Internet Explorer（以下、本稿ではIEと略記する）があるが、他のブラウザ製品がW3Cと協調して足並みを揃えて実装している機能や技術に背を向けて独自路線を選択することがあり、技術情報を得にくく、また、特別扱いをする必要があるケースが多い。

c その他のツール

デバッグ用のツールとして、Firebugs[vi], Fiddler[vii]を使用した。また、Webサーバとしてはapache 2.2を使用しているが、今回の調査では特にWebサーバに特殊な能力を必要としない範囲の技術を扱うこととしており、apache2.2であることに特段の意味はない。強いて言えばUnix系サーバOSでもWindowsでも動作している点が、ネットワーク接続のない環境でも動作確認等の作業ができる利点を提供してくれている。

5 主要な要素技術

5.1 ページ内容の動的な変化

HTMLによるページをロードし描画した後で何らかのタイミング（一般的にはマウス操作やフ

ォームのボタン押下）でページ内容を動的に変化させることは、Ajax流行の以前からDHTML（ダイナミックHTML）の呼称で広く使われている。DHTMLでは、

①スタイルの変更（CSS操作）
②DOM構造の変更（ノードの追加、削除、移動など）
③ノードの内容の変更（innerHTML属性の操作）

といった操作を行っている。DHTMLについては、すでにWeb上にサンプルやギャラリーを含めた豊富な情報源があり、本稿では詳細は省略し、上記③の使用例を1つ掲載しておく。

5.2 文書にオーバーレイさせるアノテーション

レイヤーの透明度を調整するなど、最近のCSSには多様な機能があるが、さまざまな試行錯誤の結果、意外なことに、従来からあるレイヤーは、絶対位置を指定して同じ領域に配置することでオーバーレイされて表示されることがわかった。以下のリストに示すような単純な方法で実現できる。ただし、この例では大きさを指定したラスター画像に文字を重ねているが、この技術を使う際には、文書や図形のレイアウトを細かく指定しないドキュメントの描画はブラウザ側に（各ユーザの指定に従って）委ねられることに留意しておく必要があることを注記しておく。

リスト1　ページの動的な変更

```
<html><head>
<title>ボタンを押すとテキストが追加される</title>
<script>
function append() {
  document.getElementById('target').innerHTML += 'こんにちは<BR>';
}
</script>
</head>
<body>
<form><input type="button" value="こんちは" onclick="append();"></form><br>
</body>
<div id="target"></div>
</html>
```

リスト2　オーバーレイ

```
<!doctype html>
<html><head><title>オーバーレイの実例</title></head><body>
  <DIV style="z-index:1; width:300px; height:200px;">
        <IMG src="P1000189.JPG" width=300 height=200></IMG>
  </DIV>
  <DIV style="z-index:2; width:300px; height:200px;
                  border-style:solid; border-width:2px;
                  position:absolute; left:30px; top:30px;"
   align="center">
  <font color="#ff0000">
                  このように<br> 2つのレイヤーを<br>
                  重ねて表示させることが<br>できる
  </font>
  </DIV>
</body></html>
```

図1　オーバーレイの結果

5.3　ユーザの意識的でない操作のイベント化

JavaScirpt のイベントハンドラの中で、最もポピュラーなものは、onClick であるが、それ以外に、マウスにかかわるものとしては、onMouseDown, onMouseUp, onMouseMove, onMouseOver, onMouseOut, といったものが登録されている。これらのイベントは、画面上のエレメントの上でマウスが何かの動き（ボタン押下、ボタン離す、マウス移動）をしたり、マウスカーソルがあるエレメントの領域に入ったり（Over）或いは出たり（Out）することで生成されるイベントである。このイベントハンドラは、各描画要素の、それぞれイベントハンドラ名と同じ名前の属性として、HTML のタグ中でも、JavaScript のメソッドによってでも指定できる。このイベントハンドラに設定した関数の中でデータ送信を行うコードを書くことで、マウスにかかわる意識的でない操作をモニターすることが可能となる。（イベントハンドラについては、特に独立した例をここでは提示しないが、次節のプログラムの中で使用している。ただし次節の例ではイベントの扱いについてはブラウザネイティブな方法ではなく Prototype.js の機能を使用している。）

5.4　ベクターグラフィックスの逐次描画

HTML は当初は書式つきテキストとリンクの記述からスタートしたものだが、その後、画像やテーブルなど記述能力を向上させてきている。ラスターグラフィックス（ビットマップ画像）が Web の普及の時期にすでに実用化されていたのに対し、ベクターグラフィックスの実用化は遅れていた。そのため、Web 上でフリーハンドの曲線などを表現するためには、Flash などのプラグインを用いて描画を行うか、或いはブラウザのネイティブな機能に限定する場合には、着色された微小な矩形のレイヤーを多数作成して絶対座標指定でページ上に並べて行くことで曲線っぽく見せるなどの代用策が用いられてきた。現在では、スケーラブルな画像データを扱う規格として SVG[viii] が W3C の規格になっており、ブラウザのネイティブな機能としての実装も進んで来ている（IE 以外）が、SVG では XML をベースとしたファイル単位での画像のやりとりをするものであり、リアルタイムの逐次描画には向かない。JavaScript からコールする形のベクターグラフィックス機能としては、各ブラウザに固有の描画機能をブラウザに応じて呼び出すことでクロスブラウザな機能を提供する JavaScript ライブラリがいくつか（Dojo.gfx[ix]、wz_jsgraphics.js[x]、WebFX[xi] など）公開され使われてきているが、本研究では前述の理由からブラウザネイティブな技術に焦点をあてることとし、HTML 5[xii] で規格に含まれている、Canvas 要素に着目することとした。HTML 5 は、本稿執筆時点ではまだ規格策定中のものではあるが、Canvas については、規格化を先取りする形でいくつかのブラウザにすでに実装されている。これを使って、JavaScript プログラムでコントロールしつつ、キャンバス上に図形要素を逐次配置していくことができる。以下のリストにはマウスを使ったフリーハンド描画を行うページの実現例を示す。

リスト３ Canvas の利用

```
<html><head>
<script type="text/javascript" src="..¥jsPrimer¥prototype.js"></script>
<script>
  var sx, sy, ctx;
  function init() {
          var div = document.getElementById('sample');
          var canvas = document.createElement('CANVAS');
          div.appendChild(canvas);
          if(document.uniqueID) {
                  canvas = G_vmlCanvasManager.initElement(canvas);
          }
          ctx = canvas.getContext('2d');
          $(document).observe('mousedown', mouseDown);
  }
  function mouseDown(e) {
          sx=e.pointerX();   sy=e.pointerY();
          $(document).observe('mousemove', mouseMove);
          $(document).observe('mouseup',   mouseUp);
}
function mouseMove(e) {
          var x=e.pointerX(), y=e.pointerY();
          ctx.beginPath();
          ctx.moveTo(sx, sy);
          ctx.lineTo(x, y);
          ctx.stroke();
          sx=x; sy=y;
}
function mouseUp() {
          $(document).stopObserving('mousemove', mouseMove);
          $(document).stopObserving('mouseup', mouseUp);
}
</script>
</head>
<body onload="init();">
<div id="sample"></div>
</body></html>
```

図２　フリーハンド描写

5.5　連続的な非同期通信

Web 技術において非同期通信という用語は、従来の通信技術用語として使われる場合とはやや意味がずれており、送信・受信のタイミングを合わせる合わせないの話ではなく、リクエストの送信→ドキュメントのデータの受信→描画、といったブラウザ内でのサイクルに同期しない HTTP での通信という意味だと捉えられている。この技術により、ページに表示するべき内容に変化があったときにその都度ページ全体をロードしなおす必要がなくなり、利用者を通信量の増大、不必要な待ち時間や画面のちらつきといった問題から解放することとなった。この非同期通信は Ajax の主

役となる技術であり、すでに様々なアプリケーションの普及、ライブラリや技術情報の提供が進んでいる。ユーザの（ブラウザ上での）操作を契機として通信を発生させる利用法や、setTimer()関数などによりブラウザ側で一定の待ち時間を作った上で通信を繰り返し発生させる利用法（後述のperiodical pollingパターンに使われる）についてはすでに既知のものとして扱い、本節では、次節の技術の準備として、ユーザの操作にかかわらず、またブラウザ側での待ち時間を用いずに、継続的に通信を発生させる方法について述べる。

5.5.1 IFRAME

JavaScriptで非同期通信を行うための部品として、IFRAME要素と、XMLHttpRequestオブジェクト（以下、XHRと略記することがある）の2つが広く使われている。IFRAMEは、元来はインラインフレームの意味であり、文書の中に独立したHTMLドキュメントをインラインで埋め込むためのタグであるが、このタグの裏技的な用法として、これを非可視にした上で、src属性に呼びだすべきURIを設定することで、HTTPの非同期通信が起動できる。インラインフレームからその外側の親ドキュメントのDOMを操作することも可能であり、また、インラインフレームに送られたドキュメントに含まれるスクリプトはその場で実現されることから、再度、通信を起動させるプログラム（src属性への代入）をサーバ側スクリプトから送出すれば、通信を無限に繰り返すことができる。簡単に言えば、サーバからスクリプトを送出し、「すぐにもう一度、俺を呼べ」とブラウザに伝える方法である。これを待ち時間なしで行うと、ネットやCPUへの負荷が大きくなりすぎるため、リスト4の例ではサーバ側スクリプトで1秒の待ち時間を入れている。印刷媒体でこのページの動作をデモするのは困難であるが、このページでは、（サーバ側での）日付と時刻を表す文字列が、1秒+α（様々なタイムラグが加わる）の間隔で1行ずつブラウザ画面に追加表示されて行く。

リスト4 連続的な非同期通信（IFRAME）

クライアント側

```
<!doctype html>
<html><head><title></title></head>
<body>
<iframe id="bb" style="visibility:hidden;"
    src="/cgi-bin/ajaxTest/dateStringToTargetAndRepeat.rb">
</iframe>
<div id="target" style=""></div>
</body></html>  </DIV>
</body></html>
```

サーバ側（dateStringToTargetAndRepeat.rb）

```
#!/usr/bin/ruby
require 'erb'
ERB.new(DATA.read).run(binding)
__END__
Content-type: text/html

<% sleep 1 %>
<script>
  top.document.getElementById("target").innerHTML+= "<%=Time.now.to_s%><br />";
  top.document.getElementById("bb").src="/cgi-bin/ajaxTest/dateStringToTargetAndRepeat.rb ";
</script>'
```

5.5.2 XHR

一方、XHRは、能動的な通信の主体になるべく作られたオブジェクトであり、HTTPによる通信の状態の変化や終了コードなど、様々なケースに対応すべく作られている。その反面、通信を実行するコードは以下のリストのようにやや複雑なものになる。なお、このオブジェクトはIEでは実装されておらず、IEでは同様の機能を持つ別のオブジェクトを用いることになる。下のリストでは、ほんのささやかなクロスブラウザ対応のみ行っている。XHRは、元来の意味での「非同期通信」を行っており、XHRオブジェクトを準備し、send()関数によりリクエストを送信した後はブラウザのエンジンはその通信の終了を待たずに他の処理に移行し、通信の状態に変化があった時に、予めonreadystatechange属性に設定されている関数（リストではonRecieve()）が呼ばれ、その変化した状態（readyState属性にセットされている）に応じた処理を行うというのがプログラムの考え方である。その状態は0，1，2，3，4と順次変化していき、通信が完了した時に値4となるので、4の時にレスポンスとして受信したテキストの処理を行えばいい。下のリストでは、その後すぐさま次の通信を起動することで連続的な通信を実現し、リスト4と同等の機能を実現している。

リスト5　連続的な非同期通信（XMLHttmReqest）

クライアント側

```
<!doctype html>
<html><head><title></title>
</head>
<body onload="sendReq();">
<script>
var xhr=false;
function newXHR() {
        if(typeof ActiveXObject != "undefined"){
          try {
                xhr = new ActiveXObject("Microsoft.XMLHTTP");
          } catch (e) { xhr = false; }
        } else if (typeof XMLHttpRequest != "undefined") {
                xhr = new XMLHttpRequest();
        }
}
function onRecieve() {
        if (xhr.readyState == 4 && xhr.status == 200) {
                document.getElementById('target').innerHTML += xhr.responseText+'<br>';
                sendReq();
        }
}
function sendReq(){
        newXHR();
        xhr.open('GET', '/cgi-bin/ajaxTest/dateStringAfterSleep.rb', 'True');
        xhr.onreadystatechange = onRecieve;
        xhr.send(null);
}
</script>
<div id="target" style=""></div>
</body>
</html>
```

サーバ側（dateStringAfterSleep.rb）

```
#!/usr/bin/ruby
puts 'Content-type: text/plain; charset=SHIFT_JIS';puts
sleep 5
puts Time.now.to_s
```

5.5.3 単一コネクションによる連続送信と、方式の選択

IFRAMEは、＜SCRIPT＞タグで包まれて受信したスクリプトを、受信するその都度すぐに実行しており（ここでは実行例の提示は省略したが）、またXHRでは、上記のreadyStateが4になる前に、3の状態が存在し、受信が完了せずに一部のデータを受信した状態を示している。受信したデータ長が変化する毎にこのonreadystatechangeに設定された関数が呼ばれるので、以下に示すように、単一のHTTPコネクションを長時間つなぎっぱなしにして、サーバ側プログラムからは一定の時間間隔を置いてコンテンツを送信し、ブラウザはそれを受信する都度、表示などの処理を行う、という方式で、ストリーミングに相当する通信がHTTP上で可能となる。この方式は、原理自体は非常に単純でわかりやすいものである。ただし、responseTextに設定される受信文字列は、状態2から3を経て4に遷移する過程で単調増大のみ行い、スクリプトからこの文字列を参照してもその参照した部分は残る（スクリプト側からresponseTextの変更もできない）ため、この文字列のうちどこまでがすでに処理済みのものであるかを（データ長等の数値により）スクリプトの側で管理した上で、未処理の部分のみを扱うようなプログラムを作る必要があり、この部分のコーディングは複雑になる。また、原理的には長時間の繋ぎっぱなしが可能だとしても、何らかの理由で（Webサーバでのタイムアウト、下位層での通信路切断など）でコネクションが切れることはあるので、それを検出し再接続する処理を準備することはどのみち必要になる。それを考慮すると、本項で示すようにHTTPコネクションの連続使用は行わずに、1かたまりのデータを送信する毎にコネクションをその都度完結させる方式の方がトータルでは実現が容易であるように思われる。したがって、次節の実装の際には、この長期的なHTTPによる方式は採用しないこととした。また、前2項で述べたIFRAMEとXHRを比較した際に、XHRの方がコードは若干複雑にはなるものの、プログラムからその挙動を自在に制御できる自由度の点で分があり、アプリケーションが高度化してページ内での非同期通信路を複数個使用することになった時を想定すると問題が少ないことが考えられるため、XHRでの実現を採用する。

リスト6 単一のHTTPコネクションによる連続的な非同期通信（XMLHttmReqest）

クライアント側

```
<!doctype html>
<html><head><title></title></head>
<body onload="sendReq();">
<script>
var xhr=false;
function newXHR() {
        if(typeof ActiveXObject != "undefined"){
          try {
                xhr = new ActiveXObject("Microsoft.XMLHTTP");
          } catch (e) { xhr = false; }
        } else if (typeof XMLHttpRequest != "undefined") {
                xhr = new XMLHttpRequest();
        }
}
function onRecieve() {
        if (xhr.readyState == 2 ||xhr.readyState == 3)
                document.getElementById('target').innerHTML +=
                        "2 ("+xhr.status+") "+xhr.responseText+'<br>';
        else if (xhr.readyState == 4) {
                document.getElementById('target').innerHTML +=
                        "4 ("+xhr.status+") "+xhr.responseText+'<br>';
```

```
        }
}
function sendReq(){
        newXHR();
        xhr.open('GET', '/cgi-bin/ajaxTest/dateStringRepeatedly.rb', 'True');
        xhr.onreadystatechange = onRecieve;
        xhr.send(null);
}
</script>
<div id="target" style=""></div>
</body></html>
```

サーバ側（dateStringRepeatedly.rb）

```
#!/usr/bin/ruby
puts 'Content-type: text/plain; charset=SHIFT_JIS';puts
while true do
        sleep 5
        puts Time.now.to_s
        STDOUT.flush
end
```

5.6　サーバ主導での送信

前節までは、ブラウザとサーバの関係について述べてきたが、ここでは、先生のPCと学生のPCの関係を考えたい。先生のPCで入力したコンテンツ（文字や図）を、どうやれば各学生のPCにリアルタイムで伝達できるか、である。これまでの考察から、以下のような状態を想定することができる。

- 学生はPC上でブラウザを開き、先生の指定したページを開いている。
- そのページにはXHRがこっそり仕込んであり、XHRが前節のような方法で常時サーバに対してHTTPコネクションを（途切れては再接続しつつ）維持している。
- HTTPで要求するURIの対象は、この場面では当然、ファイルではなく、ブラウザを経てCGI等の方式で呼び出されるプログラムである（元来、CGIはその呼び出しの方式を表す単語だが、世間で流布している言い回しを借りて、ここではそのプログラムをCGIと呼んでおく）。
- CGIは、前節のように一定時間sleepするのではなく、先生のPCからの何らかの通知があるのを待ってブロックしている。
- やがてその通知が届けば、CGIはブロックを解除して、ブラウザに対して通知されたコンテンツを送信して終了する。
- しかしブラウザに仕込んだスクリプトが（そのコンテンツを処理するのと並行して）即座にXHRによるHTTPコネクションを再接続する。

すると、ここでの課題は、CGIが通知をどのように受け取るか、ということになる。以下のような方法が考えられる。

① プロセスIDをどこかに登録しておき、シグナルを受け取る
② ソケットを開いて待機（Listen）し、接続を受け付ける
③ 能動的にどこかに問い合わせをしに行き、レスポンスが返ってきたことをもって通知とする。

ここでは③の考え方をとることとする。CGIは、何度も呼びなおしが行われ、いわば代替わりを絶え間なく繰り返している存在であるため、①にせよ②にせよ、そのプロセスIDやソケットとして、その存在を管理する何らかの手立てが必要である。いずれにしても、Webという仕組みに元来から欠落している、HTTPコネクション相互間の連続性、に代わるものを、別の仕組みにして用意する必要があるということである。そこで、先生PCとCGIとの間の情報伝達を仲介する、永続的なプロセスを1つ用意することにする。先生PCとその永続的プロセスの間、および、CGIと永続的プロセスの間の通信には、drubyを用いる

こととした。drubyではオブジェクトを作成し（ここではNotifierクラスと名付けておく）、そのオブジェクトに対するメソッド呼び出しを他のプロセス（同一マシン内でもリモートマシンからでも）から行うことができる。したがって、Notifierクラスの実体を動作させるこの永続的プロセスは、Webサーバと同一サーバ機で動作させてもいいし、他のマシンでもいい。ただし、drubyでの通信はHTTPとは別のプロトコルであり、他のマシンで動作させた時にWebサーバとの間のネットワークの管理方針によっては動作しないこともあり得るが、それでもこの方法は、比較的サーバの置き場所を選ばない、自由度の高いやり方だと言えるだろう。

さて、Notifierには、自分自身のコンストラクタinitialize()以外に、putMsg()とlisten()の2つのパブリックなメソッドを用意することとする。前者は送信者（先生PC）から呼び出され、後者は各CGIから呼び出されるものである。メソッドlistenは、呼び出されると、しかるべき初期処理のあと、待ち状態に入り、ブロック状態が解除されると、通知された文字列をメソッドの値として返しながら、メソッドを正常終了する。するとそのブロック状態をどのように実現するか。これでは前章の課題が、CGIからこのlisten()メソッドに移されてきただけである。しかし今度は、1つのプログラムの中だけの課題である。スレッドを扱うことができるプログラム言語には、その機能は何らかの形で用意されており、rubyではCondition Variable（以下、CVと書く）というオブジェクトがある。CVを1つ用意し、listen()ではそれをwait()（＝解放されるのを待つ）し、putmsg()ではそれを解除させるためsignal()メソッドを呼ぶ。このCV操作はクリティカルな操作であるため、相互排他ロックを実現するクラスMutexのオブジェクトを用いて、処理の同時重複実行を避ける。（この実装例は、次節における変更点が少ないため省略する）。

5.7　一斉同報

前節の技術を利用して、送信クライアントから送られたコンテンツを複数の受講者のWeb端末にリアルタイムで同報通信する方法について調査・検討し、簡便な実証プログラムを作成した。なお、1つのCVは、複数のスレッドが同時にwait()することができる。複数のスレッドによってwait()されているCVに対して、signal()メソッドを呼ぶと、wait()しているスレッドのうちの1つをブロック解除する。Signal()メソッドの代わりにbroadcast()メソッドを呼ぶことにより、wait()しているスレッド全部を一斉に解除できる。以下のようなプログラムにより基本的なデータの流れが実現できることになる。

リスト7　一斉同報

Notifierプロセス

```
require 'thread'
require 'drb/drb'

class Notifier
  def initialize
        @mutex = Mutex.new
        @cv = ConditionVariable.new
        @c=[]
        @msg = ""
  end
  # from sender
  def putmsg(msg)
        p @msg=msg
        sz=@c.size
        @mutex.synchronize {
          @cv.broadcast
```

```
        }
        "sent to #{sz} reciever: #{msg}"
  end
  # from receiver
  def listen(peer)
        @c.push peer
        p peer
        @mutex.synchronize {
          @cv.wait(@mutex)
        }
        @c.delete peer
        @msg
  end
end

DRb.start_service('druby://:9999', Notifier.new)
puts DRb.uri
DRb.thread.join
```

送信者側プログラム

```
#!/usr/bin/ruby
require "drb/drb"
serverURI=ARGV.shift || 'druby://localhost:9999'
DRb.start_service
server = DRbObject.new(nil, serverURI)
while gets do
  puts server.putmsg($_)
end
```

CUI で動作する受信側プログラム

```
#!/usr/bin/ruby
require 'drb/drb'
serverURI=ARGV.shift || 'druby://localhost:9999'
DRb.start_service
server = DRbObject.new(nil, serverURI)
while true do
        puts server.listen(self)
end
```

CGI

```
#!/usr/bin/ruby
require 'drb/drb'
puts 'Content-type: text/html; charset=SHIFT_JIS';puts
serverURI=ARGV.shift || 'druby://localhost:9999'
DRb.start_service
server = DRbObject.new(nil, serverURI)
puts server.listen(self)
```

6　まとめ

ここに報告した技術を組み合わせることで、受講者の環境をWeb上に限定しても、要求を満たすソフトウェア開発を進めて行ける見通しがついた。現在、この開発を漸次進めているところである。今後、その開発を進行させ、ある程度の安定動作を得た時点で、実際の教室での試運用をすすめていきたいと考えている。

ここで積み残している技術的課題としては、ソフトウェアの複雑性への対応の問題がある。複雑性の問題に対応しやすい道具立てとしてオブジェクト指向のいわゆる軽量言語の1つであるrubyを選んでいるが、それでもWebアプリケーショ

ンは複雑なものになりやすい。これについては今後さらに調査を進めていく必要があると考えている。

参考文献

〔1〕 平岡信之「NUANCEの思想」長野大学紀要 第16巻第4号，1995年

参考URL

i http : //www.realvnc.com/
ii http : //www.kegel.com/c 10 k.html
iii http : //www.ecmascript.org/
iv http : //www.openspc 2.org/JavaScript/Ajax/Library/index.html
v http : //www.ruby-lang.org/ja/
vi http : //getfirebug.com/jp.html
vii http : //www.fiddler 2.com/fiddler 2/
viii http : //www.w 3.org/Graphics/SVG/
ix http : //dojotoolkit.org/projects/dojox
x http : //www.walterzorn.com/
xi http : //webfx.eae.net/dhtml/chart/usage.html
xii http : //dev.w 3.org/html 5/spec/Overview.html

注

1）Asyncronus Javascirpt＋XMLとされている。
2）規格の名前としてはECMAScriptと呼ぶのが正しいが、JavaScriptのほうが通りがいいので、本稿では後者の名前で呼んでおく。