Journal of National Fisheries University 59（4） 233－243（2011）

データ分析用簡易言語Analyze実装系のクライアント・サーバ化について

楫取和明*†，青木邦匡*，瓜倉茂*

On the Client-server Implimentation for Analyze: a simple language for data analysis

Kazuaki Kajitori*, Shigeru Urikura*, Kunimasa Aoki*

Abstract : It has been too time-consuming especially in classrooms to install and update the softwares for the stand-alone Analyze system we have developed[1]. So, we consider to develop client-server（C/S） versions of Analyze system to reduce the maintenance load. We implement CGI-C/S version and RPC-C/S version of Analyze and evaluate them to check whether they are indeed better Analyze systems. The result is that they have some critical problems but those drawbacks can be lightened. Among others they fit to small classrooms and we pursue other possibilities of C/S Analyze.

ASFA keywords : Analysis, Analytical techniques, Data, Education, Client-server

はじめに

データ分析用簡易言語Analyze言語とその実行システム（Analyzeシステムと呼ぶことにする）[1]は，いろいろなデータ分析用のアプリケーションを統一的で単純な文法で使えるようにしたものである。

```
例：  apply pca of R to 'data.txt'
```

これは，フリーの統計アプリケーションである**R**[6]の主成分分析（**p**rincipal **c**omponent **a**nalysis）機能をデータファイル 'data.txt' にデフォルトのオプションで適用するAnalyze言語の文である。

Analyzeシステムでは，必要なすべてのプログラムをAnalyzeシステムを使用する各パソコンにインストールするようにしていた。このようなスタンドアローンのシステムにしたのは，とりあえず開発が簡単であったからである。開発のシンプルさはユーザレベルでの開発においては重要である。また分析アプリケーションをローカルにインストールして使っていたことからの自然な帰結でもあった。

しかしながら，使ってみるとAnalyzeシステムに必要なソフトウェアおよびそのアップデートをインストールする手間の問題が意外に大きなものであることが明らかになってきた。

特に，パソコン教室の計百数十台とか多数のパソコンにインストールする手間の問題が大きい。各パソコンに，最低数種類の分析アプリケーション，Perlの処理系とそのライブラリ，Analyzeシステムをインストールし，バージョンアップがあれば更新しなければならない。一斉にインストール・更新する方法と機会がないわけではないが，一つの科目の必要があるごとにやっている余裕はない。また，個人のパソコンにインストールする場合でも，マシンやシステムの更新時にすばやくAnalyzeシステムをインストールし直すことが難しく，新しい環境で早く作業を再開したいときに大きな負担になる（Analyzeシステムに限らずこのためにアプリケーションを使わなくなってしまうことは

2010年11月15日受付．Received November 15, 2010.

＊ 水産大学校水産流通経営学科（Department of Fisheries Distribution and Management）
† 別刷り請求先（corresponding author）: kajitori@fish-u.ac.jp

少なくない)。

この問題については実装を見直すことで対処できるのではと考えられた。すなわち，Analyzeシステムの主要部分はサーバに置き，クライアントはユーザインターフェイスとサーバとのやりとりのみを担うクライアントサーバシステムにすることである。「クライアントサーバ」という用語はいろいろな意味合いで使われるので，本論ではこの意味で使うことをお断りしておく。この意味でのクライアントサーバ方式を今後簡単にC/Sと表現することにする。

C/S方式にすれば，インストールとアップデートのほとんどはサーバのみで済むのでクライアントにおける管理は格段に楽になる。

しかしながら，C/Sシステムも，スタンドアローンシステムと比較して長所ばかりでなく短所もある。本論では，C/S化したAnalyzeのシステムを２種類実装し，その実装にもとづいて，C/S化されたAnalyzeシステムのパフォーマンスを検証しC/S化の総合的な評価をする。

Analyzeシステムの開発は，Perl，Linuxなど，1970年代以降開発され公開されてきたさまざまなテクノロジーに負っている。これらのテクノロジーがあってこそ，ユーザレベルでユーザニーズに応じたシステムの開発が可能になっている。AnalyzeシステムのC/S化についても，HTTPやXML-RPCのプロトコルとそれらをサポートするPerlのライブラリなどオープンテクノロジーのおかげで実現できたものであることを記しておく。

C/S Analyzeシステムの実装

概　要

スタンドアローンAnalyzeシステムはスクリプト言語Perlで書かれ，ほとんど同じコードでLinuxとWindows上で稼動した[1]。C/S AnalyzeシステムもPerlで開発することにする。C/S Analyzeシステムの構成要素はつぎのようになる（Fig 1も参照のこと)。

- ユーザが書いたAnalyze文をサーバに送り，返された結果を表示（など）するAnalyzeクライアント
- クライアントから送られたAnalyze文を実行して結果を返すAnalyzeサーバ
- （Analyzeサーバ内）Analyze文をPerlのプログラムに翻訳するプログラム
- （Analyzeサーバ内）分析アプリケーションとそのドライバ（ドライバとは，Analyzeシステムから分析アプリケーションを使えるようにするソフトウェアのことである）

最初の分析アプリケーションとドライバについては，サーバ用のものだけ用意すればよいので，スタンドアローンのAnalyzeシステムと比べ実装面での有利さになる。

パーサはスタンドアローン版と本質的に変わらない。

スタンドアローン版と比べて複雑になるのは，ユーザからAnalyze文を受け取って結果を返すプログラムがサーバとクライアント部分に別れその間に通信が入ることである。

C/S化の仕組みとしては，プラットフォーム非依存であること，かつ実装がなるべく簡単であることが望ましい。そこでサーバ部とクライアント部間の通信には，C/Sアプリケーション開発の裾野を拡げてきたCGI（Common Gateway Interface）[10] とRPC（Remote Procedure Call）[11] による実装を試みる。CGIは動的なウェブページを作成するのに使われている仕組みである。RPCはサーバに実装されているルーチンをクライアントが直接呼び出す形で使うためのプロトコル（通信手順）である。

クライアントプログラムは，CGI版もRPC版もスタンドアローン版と同様にAnalyze文を直接入力するコマンドラインシェル式とする。

CGIアプリケーションではクライアントプログラムをIEやFirefoxなどのウェブブラウザで済ませば，クライアント側では何も新規にインストールしなくて済む。しかし

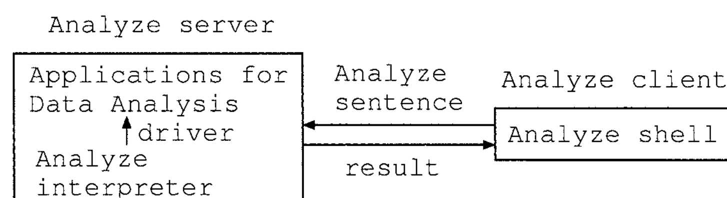

Fig. 1. The C/S Analyze system

ウェブブラウザではコマンド補完やヒストリなどのシェルの機能の実現は難しい。また，ウェブブラウザはセキュリティ上，ユーザの確認なしにデータをクライアント側に保存することはできにくくなっている。これではヒストリ，ログ，分析結果を保存することも面倒になる。したがって，CGI版Analyzeシステムでもクライアントプログラムは独自に開発することにする。

クライアントシェルの骨格はCGI版でもRPC版でも同じで以下のようになる。ReadLineライブラリによってコマンドラインシェルを実現するものとなっている。

```
use Term::ReadLine;
my $term = new Term::ReadLine 'Analyze shell';
#コマンドラインシェル機能を実現する
…
open LOG,">ana_log"; close LOG;# 古いana_log
をフラッシュ(空ファイルにする)
…(ここで，サーバにアクセスして$session_idを得る)
while(defined($sentence=$term->readline('Analyze> '))){
    if($sentence=~/exit/i){exit 0;}
    open LOG,">>ana_log";
    print LOG ">Analyze> $sentence\n";
    …(ここに，Analyze文$sentenceに対するサー
    バからの結果を得るコードが書かれる)
    …(ここに，結果をその属性に依存して表示する
    コードが書かれる)
    …
}# while loop 終わり
```

サーバプログラムは，CGI版とRPC版では骨格が異なるが，CGI::Sessionモジュール[19]を用いてクライアントプログラムの立ち上げごとにsession_idを発行してサーバにおける一時データを区別する点は同じである。

また，サーバプログラムの主要部はAnalyze文を解釈して実行する部分であり，ここはスタンドアローン版，CGI版，RPC版を通じて共通で，Parse::Yappモジュールによって生成したパーサプログラムparse. pmのオブジェクトを生成してAnalyze文をPerlのプログラムに変換し，evalで実行する。Analyze言語の文法は非常に単純なのでパーサプログラムについては省略する。

CGI版の通信部分

CGIはウェブアプリケーションでお馴染みの仕組みであり，著者らも開発の経験が多い。CGIではHTTP（HyperText Transfer Protocol）[12]という簡単なプロトコルを用いて通信が行われる。クライアントが送ったリクエストは，サーバ上でアプリケーションによって処理され結果がクライアントに返される。サーバ上でHTTPでのやりとりを司るのは通常ウェブサーバ（HTTPサーバ）と呼ばれるソフトウェアでここでもウェブサーバの使用を前提とする。

HTTP通信をフィルタリングすることはセキュリティ上も少ないから，クライアントが外部からアクセスしやすい利点もあろう。

まずクライアントプログラムである。PerlモジュールのLWP::UserAgentとHTTP::Requestを使用するとformデータをウェブサーバに送り結果を受け取るプログラムが以下のように書ける。

```
my $ua = LWP::UserAgent->new;
…
while(...){# 上述コマンドdrivenループ
    …
    my $request_up = POST(
      'http://10.0.0.2/perl/server_analyze2.pl/up',
      Content_Type => 'form-data',
      Content => { upfile => ["./$datafile"], },
    );
    my $response_up=$ua->request($request_up);
    my $request_ana = POST(
      'http://10.0.0.2/perl/server_analyze2.pl/response',
      Content => { analyze => $sentence,},
    );
    my $response_ana=$ua->request($request_ana);
    …(結果$response_anaを表示)
}# コマンドdrivenループ終わり
```

主要部は二つのPOSTコードで，最初のものがデータファイルをサーバにアップするもの，次がそのデータに対して分析を行うコードである。

それに対するCGI版Analyzeサーバ側のコードの通信部分を以下に示す。HTTPによる通信はウェブサーバが担うのでHTTP通信のコードは含まない。クライアントからのリクエストはウェブサーバが受け取りこのサーバプログラムに渡され結果をウェブサーバがクライアントに返す。

```
…
my $query = new CGI;
my $path_info = $query->path_info;
…
my $id=$query->param('session_id');
if($path_info=~/up/){
    print $query->header(-type=>'multipart/
      form-data', -charset=>'utf8');
    my $fh = $query->upload('upfile');
    …(エラーハンドリング)
    copy($fh,"/var/www/html/up/$id.datafile");
    return;
}
if($path_info=~/response/){
    print $query->header(-charset=>'utf8');
    my $sentence=$query->param('analyze');
    my $server_datafile_path="/var/www/html/
      up/$id.datafile";
    # to句のデータ名もこのサーバでのファイル名
    に変えておく:
    $sentence=~s/(\s+to\s+')\S*('\s*)/
      $1$server_datafile_path$2/;
    …(Analyze文$sentence解釈実行コードがここに)
    print $result->{'default'}; # can't return hash
}
```

クライアントのリクエストにおけるpath情報によってuploadかapply文の実行かに分岐するようにしている。

RPC版の通信部分

RPCならAnalyzeシェルからサーバ側のルーチンを直接呼んで結果を得ることができるところが特徴で通信コードの少ないシームレスな（通信相手を意識しない）コードが書ける。

今回のRPCによる実装に当たっては，RPCの仕組みの一つであるXML-RPC[14] を使うことにする。XML-RPCとは，RPCプロトコルの一種であり，転送機構にCGIと同様にHTTPを採用し，情報はXML[13] にエンコードしてやりとりするものである。

すなわち，クライアントは呼び出したいサーバのルーチンとそのルーチンに渡す引数をXMLにエンコードしてHTTP経由でサーバに送る。するとサーバは，そのXMLコードをXMLパーサでデコードしてそれにしたがってサーバのルーチンを動かし，そのリターン値をXMLにエンコードしてHTTP経由でクライアントに返す。クライアントは受け取ったXMLコードをXMLパーサでデコードして結果とする，という仕組みである。HTTPを使用することからCGIと同じようにウェブサーバ（HTTPサーバ）を使用することもできるが，ここでは独自のサーバプログラムを書くことにする。

XMLはデータをテキストデータでエンコードするための標準である。標準であるから異なるプラットフォーム間のやりとりに使われるのである。

XML-RPCはインターネットのFirewall環境で使えるようにCGIを超えた機能を持たない[15] とされているように単純なプロトコルであり，CGIと同様に広い環境で使うことができる。

我々のAnalyzeシステムではPerlを使用していることから，XML-RPCのPerl実装でLinuxでもWindowsでも使えるFrontier-RPCモジュール[16] を使用する。

クライアントシェルの骨格は前述のとおりで，ここではRPC版特有のサーバのメソッドをコールして結果を受け取る部分を示す：

```
my $server = Frontier::Client->new(url =>
  'http://10.0.0.2:7070/RPC2');
…
while(...){# 上述コマンドdrivenループ
    my $result=$server->call('server_analyze',
      $session_id,$sentence,$datastr);
    …($resultを表示)
}# コマンドdrivenループ終わり
```

ユーザが入力したAnalyze文をサーバ側の'server_analyze'メソッドを呼び出して処理し，結果$resultはハッシュ（リファレンス）として受け取っている。ここで，データはファイルから読み込んで文字列$datastrの形でサーバのメソッドに渡しているので，ファイル送信のコードは必要ない。RPCならではの透過的な記述であり，ファイル送信が済んでから分析が始まる保証があるかを心配する必要がない。

これに対するRPC版Analyzeサーバ側のコードの通信部分を以下に示す。

```
my $methods = {'session_start' => \&session_start,
               'upload_datafile' => \&upload_datafile,
               'server_analyze' => \&server_analyze};
Frontier::Daemon->new(LocalPort => 7070, methods => $methods)
               or die "Couldn't start HTTP server: $!";
```

$methodsにあるような各ルーチンを定義しておいて，ポート7070でHTTPサーバを立ち上げクライアント側からのルーチンコールを待ち受けるのである。

C/S Analyzeシステムのインストールについて

C/S Analyzeシステムでは，種々の分析用アプリケーションとそのドライバ，Analyzeシステムのパーサ，それにドライバとパーサを動かすプログラムはすべてサーバに置くので，クライアントに必要なプログラムは，Perlとそのモジュール，Analyzeシェルだけである。

CGI版では必要なPerlモジュールは標準でPerlとともにインストールされるので，新たに加えるものはない。RPC版でもXML-RPCモジュール（Frontier-RPC）を別にインストールするだけでよい。

したがって，クライアントでのバージョンアップもAnalyzeシェルだけであるから，ユーザにダウンロードしてもらうことで済ますこともできる。

スタンドアローン版で負担だったインストール（バージョンアップ含む）作業は格段に楽になった。

C/S版開発の難度について

Analyzeシステムはユーザレベルでの開発を意図するもので，開発の簡単さが重要である。C/S化によって開発の難度は増すがそれがどのぐらいかを測るのは難しい。ここではプログラムの行数を比べてみる。スタンドアローン版とC/S版のPerlプログラムの行数を比べれば以下のようになる。この表は次章のベンチマークに必要な部分で比べている。また，スタンドアローン版のシェルはclientと見ている。

Table 1. The number of lines of the programs of Stand-alone Analyze and C/S Analyze systems

	parser	client	server	total
Stand-alone Analyze	94	27	0	121
CGI C/S Analyze	80	73	52	205
RPC C/S Analyze	80	52	40	172

行数からいってもC/S版はスタンドアローン版から増えており開発の難度は増している。しかし，200行近辺の行数で収まっているのでまだまだ簡易スクリプトの範囲といえよう。したがってC/S Analyzeシステムは，簡易なプログラミングで多くの分析アプリケーションを活用するというAnalyzeシステムの当初の目的になお適っている。

通信コードが入ってこの程度の増加で納まっているのはネットワークがらみの部分にPerlの豊富なモジュールライブラリを駆使しているからである。

しかしCGI版ではプロトコルはHTTPでXML-RPCに比べて単純であり，その分いろいろなコードを自前で書かなくてはならない。たとえば，データファイルはRPCではクライアントで読み込んで変数でサーバメソッドに渡せるが，CGIではファイル転送しなくてはならない。またXML-RPCでは結果をハッシュでクライアントに返すことができるが，CGIでは無理なので結果の属性によって処理方法をクライアントで変えるのが面倒である。

このように開発の手間に関してはRPC版の方が有利である。

C/S Analyzeシステムの実行環境について

スタンドアローン版Analyzeシステムは，Linux上でもWindows上でも動くことは確認してある[1]。このたびC/S版の開発に当たって，C/S版（CGI版およびRPC版）も，Linux上でもWindows上でもほとんど同じコードで動くことが確認できた。

Windows上のPerlとしては，ActivePerl[4]とStrawberry Perl[5]があるが，どちらのPerlを使ってもスタンドアローン版，C/S版ともまったく同じコードで動いた。

詳しい実行環境とレスポンスに関しては次節で述べるが，レスポンスを欲張らなければAnalyzeシステムの単純さ故に移植性も十分であることが確認できた。

C/S Analyzeシステムのレスポンスの検証

この章では，C/S版のレスポンスについて検証する。

レスポンスの計測には，線形計算からなる分析を実行するAnalyze文を使用する。多くの数値計算が線形計算に還元されることから，システムの数値演算性能を測るベンチマークには線形計算がよく使われるからである。

使用するAnalyze文は以下である。

```
apply io_inverse of octave to 'io05b101.tab' with 'I-A'
```

このapply文を（*）で引用する。使用データはio05b101.tab（約200KB）で，総務省統計局の平成17年度産業連関表（input-output table, io table）[8]の中から投入係数表190部門表io05b101.xlsをダウンロードし，その数値データ部分をテキストに落としたものである。水産経済に関連した少し大きめのデータとして選択した。

データのスケールに対する応答を見るために，その他につぎのようなものを作成した。io05b101.tabのデータを横に二つ並べ，さらにそれを縦二つに並べると380行380列のデータができるが，その人工的データをio05b101_double.tab（約800KB）とした。同様に，570行570列データを作り，io05b101_triple.tab（約1800KB）とした。

上のapply文（*）は，io05b101.tabを投入係数行列A（次数n=190の正方行列）として，モデル（I-A）$^{-1}$の逆行列表[9]を求めるものである。このapply文を実行すれば，クライアントマシンにあるデータファイルio05b101.tabがサーバで処理され，結果の逆行列データはクライアントマシンに保存される。データを他の二つに変えればn=380，n=570の場合の逆行列が得られる。

apply文（*）の実行で必要な計算は行列の演算であり，とくにここでは逆行列の計算に時間がかかる。そこで，線形計算にAtlas[3]という高速線形計算ライブラリを活用しているoctave[7]をアプリケーションとして使用している。

時間計測は，Analyzeシステムのクライアントプログラムに，PerlのTimes::HiResモジュール[7]のgettimeofdayを使って，実行後（結果が返ってきて保存されたとき）の時刻から実行前の時刻（apply文（*）を実行するためにEnterキーを押したとき）を引いたものを記録するコードを加えて行っている。

まず，以下の実験で使用するマシンのスペックを表にしておく。

各実験においては，n=190，380，570のそれぞれの場合についてapply文（*）の実行時間を各5回計測しその平均時間を結果として記す（時間単位は秒）。

以下の実験を通じてクライアント，サーバ間はすべて100Base-TのLANでつながれている。

CGI版C/S AnalyzeシステムにおいてはHTTPサーバとしてApache 2.2（mod_perlつき）を使用する。

Windows上のPerlとしては，2種類（ActivePerlとStrawberry Perl）とも試したが，ActivePerlの方が明らかに速かったので以下ではActivePerlを使用した計測記録のみ示す。

単独のアクセスに対するレスポンス

この節においては，複数のクライアントを同時に動かさず単独のクライアントのみでapply文（*）を実行した結果を見る。

Computer 1～4を使用して，スタンドアローン版との違いや，OSでの違い，CPUでの違いを見た。結果を，Table 3に示す。ただし，サーバとクライアントが同じ場合は，自分自身にローカルループバック経由で接続しているのでその分速いことをお断りしておく。

スタンドアローン版との比較

スタンドアローン版との比較では，CGI版はスタンドアローン版と遜色ない速度を示している。n=570の場合の結果データの6MB程度ならデータのLAN転送のオーバーヘッドは少ないことがわかる。

RPC版では，データのサイズが大きくなるほどスタンドアローン版との差が顕著になる。しかし，Frontier-RPCモジュールを新しいモジュールであるRPC::XMLに変えるとCGI版よりやや遅い程度となる。（表中数値に*印がついた行。すぐ上の行のFrontier-RPCを使ってそれ以外は同じ条件の行と比べられたし。n=190の場合はわずか

Table 2. The specifications of the computers used for our experiments

name	CPU	RAM	OS	alias
Computer 1	Core i5 3.3GHz	2GB	Linux Fedora 13	Linux,i5-3.3
Computer 2	Core i5 3.3GHz	2GB	Windows XP	WinXP,i5-3.3
Computer 3	Core 2 Duo 2.1GHz	2GB	Linux Fedora 13	Linux,Duo2.1
Computer 4	Core 2 Duo 2.1GHz	2GB	Windows XP	WinXP,Duo2.1

Table 3. The execution time of various Analyze systems

Analyze system	Server	Client	$n = 190$	$n = 380$	$n = 570$
Standalone	None	Linux,i5-3.3	0.200	0.490	0.997
Standalone	None	Linux,Duo2.1	0.325	0.784	1.579
Standalone	None	WinXP,i5-3.3	0.891	1.947	3.887
Standalone	None	WinXP,Duo2.1	1.387	3.031	6.091
CGI C/S	Linux,i5-3.3	Linux,i5-3.3	0.216	0.548	1.156
CGI C/S	Linux,i5-3.3	Linux,Duo2.1	0.296	0.839	1.760
CGI C/S	Linux,i5-3.3	WinXP,i5-3.3	0.289	0.834	1.744
CGI C/S	Linux,i5-3.3	WinXP,Duo2.1	0.311	0.844	1.769
CGI C/S	Linux,Duo2.1	Linux,i5-3.3	0.443	1.132	2.328
CGI C/S	Linux,Duo2.1	Linux,Duo2.1	0.386	0.892	1.782
CGI C/S	Linux,Duo2.1	WinXP,i5-3.3	0.422	1.137	2.378
CGI C/S	Linux,Duo2.1	WinXP,Duo2.1	0.456	1.153	2.387
CGI C/S	WinXP,i5-3.3	Linux,i5-3.3	0.897	2.118	4.181
CGI C/S	WinXP,i5-3.3	Linux,Duo2.1	0.922	2.144	4.198
CGI C/S	WinXP,i5-3.3	WinXP,i5-3.3	0.844	1.894	3.678
CGI C/S	WinXP,i5-3.3	WinXP,Duo2.1	0.922	2.131	4.222
CGI C/S	WinXP,Duo2.1	Linux,i5-3.3	1.453	3.372	6.569
CGI C/S	WinXP,Duo2.1	Linux,Duo2.1	1.462	3.391	6.581
CGI C/S	WinXP,Duo2.1	WinXP,i5-3.3	1.472	3.378	6.637
CGI C/S	WinXP,Duo2.1	WinXP,Duo2.1	1.369	3.012	5.875
RPC C/S	Linux,i5-3.3	Linux,i5-3.3	0.259	1.145	3.542
RPC C/S	Linux,i5-3.3	Linux,i5-3.3	*0.288	*0.772	*1.560
RPC C/S	Linux,i5-3.3	Linux,Duo2.1	0.345	2.494	7.701
RPC C/S	Linux,i5-3.3	Linux,Duo2.1	*0.323	*0.905	*1.861
RPC C/S	Linux,i5-3.3	WinXP,i5-3.3	0.328	1.446	4.174
RPC C/S	Linux,i5-3.3	WinXP,Duo2.1	0.442	2.262	6.715
RPC C/S	Linux,Duo2.1	Linux,i5-3.3	0.460	1.670	5.316
RPC C/S	Linux,Duo2.1	Linux,i5-3.3	*0.487	*1.297	*2.636
RPC C/S	Linux,Duo2.1	Linux,Duo2.1	0.397	2.464	8.345
RPC C/S	Linux,Duo2.1	Linux,Duo2.1	*0.469	*1.263	*2.560
RPC C/S	Linux,Duo2.1	WinXP,i5-3.3	0.453	1.696	5.303
RPC C/S	Linux,Duo2.1	WinXP,Duo2.1	0.568	2.490	7.912
RPC C/S	WinXP,i5-3.3	Linux,i5-3.3	0.990	2.771	6.441
RPC C/S	WinXP,i5-3.3	Linux,Duo2.1	1.007	3.821	9.998
RPC C/S	WinXP,i5-3.3	WinXP,i5-3.3	0.937	2.496	5.777
RPC C/S	WinXP,i5-3.3	WinXP,Duo2.1	1.137	3.512	8.871
RPC C/S	WinXP,Duo2.1	Linux,i5-3.3	1.550	4.043	9.080
RPC C/S	WinXP,Duo2.1	Linux,Duo2.1	1.539	5.103	12.641
RPC C/S	WinXP,Duo2.1	WinXP,i5-3.3	1.562	4.074	9.140
RPC C/S	WinXP,Duo2.1	WinXP,Duo2.1	1.490	4.412	10.856

The values with (*) marks are obtained with RPC::XML module. Other values are obtained with Frontier-RPC module.

Table 4. The execution time of encoding and decoding of XML-RPC modules

	$n = 190$	$n = 380$	$n = 570$
Encoder of Frontier-RPC	0.005	0.023	0.052
Encoder of RPC::XML	0.005	0.024	0.053
Decoder of Frontier-RPC	0.035	0.569	2.013
Decoder of RPC::XML	0.008	0.030	0.065

にRPC::XMLの方が遅いところも見えるが，n=380，n=570の場合の違いが大きい。）

Table 4 に，Frontier-RPCのコーダとRPC::XMLのコーダのスピードの比較を載せている。これはComputer 1（Linux，i5-3.3GHz）において，io05b101.tab，io05b101_double.tab，io05b101_triple.tabの３つの投入係数表に対してoctaveが生成した３つの逆行列表の結果をFrontier-RPC，RPC::XMLのエンコーダでエンコードして，それぞれの結果をFrontier-RPC，RPC::XMLのデコーダでデコードした時間を測定している。

Frontier-RPCでもRPC::XMLでもエンコードの時間はほとんど変わらないし，エンコードの内容もほぼ同様のものである。Frontier-RPCのデコーダはRPC::XMLのデコーダに比べるとかなり時間がかかり，n=570の場合その差がトータルでのapply文（*）の実行時間の差にほぼ等しい。参考文献[2]にもあるとおりXML-RPCにおいてはXMLエンコード・デコードがオーバーヘッドになりうる（本実験の場合特にデコード）。

RPC::XMLではXMLのデコードが速くデータサイズの影響も小さいので，デコーディングのオーバーヘッドは小さくて済む。しかし，RPC::XMLは現在のところWindowsでは動かないので残念ながらFrontier-RPCを標準とせざるを得なかった。この点については「考察」でまた触れる。

OSによる違い

OS以外の条件が同じ場合でLinuxとWindowsを比べれば，Linuxの方が速い。スタンドアローンのAnalyzeシステムが，WindowsでLinuxより遅いのはWindows版octaveがデータファイルを読み込むのが遅いからである（比較計測データは省略する）。しかし，C/S Analyzeシステムではサーバのoctaveを使うので，C/S Analyzeシステムのクライアントがwindowsで遅いのは違う要因である。おそらく，Windowsサーバ，Windowsクライアントともに，Perlの実行速度がLinux上に比べ遅いのが原因であろう。

この点に関しては，Analyzeシステムのクライアントは単純で小さいので，クライアントはJavaなどWindows上でオーバヘッドのない他の言語で書くことも考えられる。CGIはもちろんRPCでもサーバとクライアントで別の言語が使えるということになっている。

CPUによる違い

スタンドアローン版では，実行時間はほぼCPUのクロックに反比例する。CGI版での実行時間は，サーバのCPUに依存するが，クライアントのCPUにはほとんど依存しない。これは，C/S Analyzeシステムの実行の主なプロセスがサーバで行われるからである。しかし，RPC版（Frontier-RPC使用の場合）での実行時間は，サーバのCPUにも依存するが，データサイズが大きくなるとクライアントのCPUにも依存する。これは，RPC版ではクライアントにおいてもXMLのエンコード・デコードが行われ，この負荷がデータのサイズに応じて大きくなるからである。

複数のアクセスに対するレスポンス

次の実験は，apply文（*）の実行を５台のクライアントから連続して行って各クライアントで実行時間を測定した。ここで，連続して行うとは，各クライアントでapply文をAnalyzeシェルに書いてあとはEnterキーを押すだけにしておいて，つぎつぎにEnterキーを押していくということである。各クライアントで，Enterキーを押してから結果が返ってきて保存されるまでの時間を計測する。連続実行の負荷をみるために負荷の高いRPC版で行う。５台すべてのEnterキーを押すのにかかる時間は２，３秒である。

クライアントマシン（4-1～4-5）はすべてComputer 4（Core 2 Duo 2.1GHz，Windows XP）と同一機種（Dell Vostro 1510）である。Windowsマシンを選んだのは，クライアントとしてもっとも多く使われると思われるからである。サーバにはComputer 1（Linux，Core i5 3.3GHz）を使った。

Table 5. The execution time of successive accesses from clients //for RPC C/S Analyze system

	$n=190$	$n=380$	$n=570$
Client Computer 4	0.442	2.262	6.715
Client Computer 4-1	0.445	2.231	7.028
Client Computer 4-2	0.506	2.547	7.834
Client Computer 4-3	0.495	2.806	9.053
Client Computer 4-4	0.509	2.981	10.241
Client Computer 4-5	0.522	2.650	11.003
Average(4-1～4-5)	0.495	2.643	9.031

結果はTable 5のようであった。

Client Computerの番号順にapply文を実行している。同じnのサイズで5回行っており，表の値はその5回の平均である。比較のためにClient Computer 4として，単独実行の時間を併記してある。

データのサイズが大きくなり各クライアントから要求された計算がサーバ上で並行して実行されるようになると，単独実行のときとの差が顕著になる。

考　察

CGI版とRPC版について

CGI版は開発が面倒であるがレスポンスは速く，RPC版は開発はしやすいがレスポンスでは不利である。

CGI版ではファイル転送とAnalyze文実行を切り離して実行している。これはともするとファイル転送が終わってないのにAnalyze文の実行が始まるという懸念を払拭できない。実際，Windows上でStrawberry Perlを使った実験では空の結果が帰ってくるということがたまに起きたことを確認している。RPC版ではそのようなことはプログラムの構造上考えにくいし，実際疑われるようなトラブルは起きていない。

RPC::XMLなどXML-RPCの新しいPerl実装が開発が現在進んでいるところなので，これらがRPC版のレスポンスの問題をWindows上でも軽減してくれるであろう。プログラムの複雑度に関してはRPC版が有利であるから，Analyzeシステムのさらなる高機能化が必要である限りその有利さは将来にわたって維持されるだろう。将来レスポンスの問題が軽減されるのであれば，仕様の単純さと開発の容易さを旨とするAnalyzeシステムにとってはRPC版の方がふさわしいように思われる。しかし，CGI版で使われているHTTP通信のモジュールは実はRPC版でもFrontier-RPCやRPC::XMLの中で使われているので，RPC版のベースの動きを理解するためにもCGI版の開発は有益であった。

データ転送とuse文の導入について

C/S Analyzeシステムでは，今日の速いLANに対するデータ転送の負荷はさほど考慮する必要はないとしても，インターネット経由でのやりとりとなると，そう頻繁に何メガ以上のデータが行き来しては負担になるであろう。前節で述べたデータ転送と分析実行の同期の問題と現状のRPC版のオーバーヘッドの問題にとってもデータのやりとりは少ない方がいい。

C/S Analyzeシステムのサーバではユーザのアクセスごとにセッションを張るからデータを前もって送ることもできる。

```
Analyze> use data 'io05b101.tab'
Analyze> apply io_inverse of octave with 'I-A';
```

このように新たにuse文を設けて，それを用いてデータを送ってからto句なしのapply文を実行するのである。一度データをアップロードしておけば，同じデータを何回も使う場合はデータ転送は一回で済むのでデータの行き来を減らすことになる。to句を省略できるという利点もある。

use句は一般に当面のデフォルトを指示するのに使える。use app octaveとすれば'of octave'を省略できるというように，to句以外にof句，with句の省略にも使えるので便利であろう。

またRPCのオーバーヘッドは，データのアップロードだけでなく分析結果のダウンロードに対しても起きる。apply文の終わりにセミコロン（;）をつけたのは，octave言語などでも採用されているように結果の表示を抑えるという意味で，結果のダウンロードをせずにサーバに蓄えておくことを意味する。

C/S Analyzeシステムの活用と課題

レスポンスの結果を見ても，C/S Analyzeシステムは，本論の実装のままでも小規模データに対してはそのまま使える。メンテナンス上のメリットを生かせば，比較的少人数の教室（30人ぐらいまで）でデータのサイズには配慮しながら使うといった使い方がひとまず最も適した使い方として考えられる。

例えば，産業連関分析では部門数は多くて数百のレベルであり（日本国の連関表の例で内生部門の基本分類表が520行×407列，東京都地域の例で内生部門の基本分類表が597行×482列），前章のベンチマークで使った逆行列表の計算が連関分析でよく使われる処理では最も重い処理なので，本論のFrontier-RPC版C/S Analyzeシステムを用いた教室での分析実行が可能である。

また，多変量解析ではよく使う主成分分析では，結果を人間が見てトップの２，３個の主成分を分析解釈するようなデータであれば項目数は多くて数十であろう。主成分を求める計算は，項目数を次元とする対称行列の固有値・固有ベクトルを求めることであり，これは逆行列を求めるより計算量はかかるが数十次元ならFrontier-RPC版C/S Analyzeシステムでまったく問題はない。

ちなみに，筆頭著者が現在水産大学校で行っている授業（履修者29人）で本論のFrontier-RPC版C/S Analyzeシステムを使い始めた。学生にとってAnalyzeシステムは初めてであるが，大きな問題もなく導入できている。学生がコマンドラインインターフェイスに大きな戸惑いを示す兆候もない。教室は，クライアントマシンとして本論の実験で使ったCore 2 Duo, 2.1GHzのWindows XPマシンとそれより遅くメモリーも少ないWindows XPマシンも混ざる環境であり，サーバには本論の実験で使ったCore i5, 3.3GHzのLinuxマシンを別の部屋で立てている。授業で扱うデータ分析としては，上述の主成分分析と産業連関分析を取り上げる予定であり現在主成分分析（最大44項目データの分析）を行っているが，レスポンスを含め問題はとくにない。産業連関分析では扱う表の部門数は200以下で済む予定なのでやはりレスポンスの問題はないものと思われる。

レスポンスの検証で使った産業連関表の分析は行列の計算を多用するものであり，低レベル（人間より計算機に近いレベル）のアプリケーションやライブラリの使用が速度の面や柔軟な計算手順が組める点で適している。産業連関分析において逆行列表の計算は，以前は計算量が多いのであらかじめ計算しておくもので部門構成を変えていちいち再計算はしないものであったが，現在の計算機と数値計算アプリケーションを使えば条件を変えて何度でも計算できる。また，計算量のかかる逆行列表を経ないで波及効果を計算することもできるし，移輸入のからむ域内需要の計算を自動化することもできる。本論の実験で使ったoctaveなどがそうした行列計算が容易にできる数値計算アプリケーションであるが，数値計算を日常的に行うわけではない一般には馴染みが薄いと思われる。このようなとき，共用のC/S版Analyzeシステムであればユーザ各人が自分でアプリケーションをインストールする必要もなく，あまり普段使わないようなアプリケーションをAnalyze言語の単純な文法で使える。

個人使用でも共用のサーバを立てて各人のメンテナンスを軽減し，Analyzeシステムの単純で統一的な方法で各種アプリケーションにアクセスできれば，そのメリットが通信のオーバーヘッドやアクセスが重なったときのレスポンスの低下というデメリットを越える場合もありうる。

Analyzeシステムの趣旨はいろいろな分析アプリケーションを簡単に使うということであるから，RPC版のレスポンスに関する弱点を克服すれば，面倒なことはサーバに一点集中させることができるC/S Analyzeシステムが有効なシーンは少なくないと思われる。

メンテナンスと開発の容易さを考慮すれば，RPC版C/S Analyzeシステムが最もAnalyze言語実装系としてふさわしいと考えられるので，以後そのレスポンスの問題点の克服を課題としたい。また現在のところAnalyzeシステムでは，回帰分析，主成分分析，サポートベクトルマシン，産業連関分析，決定木分析，ベイジアンネットワークの分析ができるが，使える分析手法も増やしていく予定である。本論の基本的なC/S Analyzeシステムの実装をさらに実用的なものにし，さまざまなシーンにおいてその有効性を検証していきたい。

参考文献

1）楫取和明（I），瓜倉茂（I），青木邦匡（I），データ分析用簡易言語の実装, 水産大学校研究報告, 58（3），191-198（2009）.

2）M.Allman, An Evaluation of XML-RPC, ACM SIGMETRICS Performance Evaluation Review, 30（4），2-11（2003）.

3）R. Clint Whaley, et.al, "Automated Empirical Optimization of Software and the ATLAS project", Parallel Computing, 27（1-2），3-35（2001）.

4）ActivePerl, http://www.activestate.com/activeperl.

5）Strawberry Perl, http://strawberryperl.com/.

6）The R Project for Statistical Computing, http://www.r-project.org/.

7）octave, http://www.gnu.org/software/octave/.

８）平成17年（2005年）産業連関表（確報），
http://www.e-stat.go.jp/SG１/estat/List.do?bid=000001019588\&cycode=０.
９）産業連関分析について，
http://www.stat.go.jp/data/io/bunseki.htm.
10）FOLDOC, Common Gateway Interface,
http://foldoc.org/Common+Gateway+Interface.
11）FOLDOC, Remote Procedure Call,
http://foldoc.org/RPC.
12）FOLDOC, Hypertext Transfer Protocol,
http://foldoc.org/HTTP.
13）FOLDOC, Extensible Markup Language,
http://foldoc.org/XML.
14）XML-RPC Home Page, http://www.xmlrpc.com/.
15）XML-RPC Specification,
http://www.xmlrpc.com/spec#update3,（2003）.
16）Frontier-RPC,
http://search.cpan.org/~kmacleod/Frontier-RPC-0.07b４/.
17）LWP::UserAgent,
http://search.cpan.org/~gaas/libwww-perl-5.837/lib/LWP/UserAgent.pm.
18）HTTP::Request,
http://search.cpan.org/~gaas/libwww-perl-5.837/lib/HTTP/Request.pm.
19）CGI::Session,
http://search.cpan.org/~markstos/CGI-Session-4.42/lib/CGI/Session.pm.
20）Time::HiRes, http://search.cpan.org/~jhi/Time-HiRes-1.9721/HiRes.pm.