The MultiHttpServer
A Parallel Pull Engine

Christoph Endres
email: Christoph.Endres@dfki.de

April 1999
Deutsches Forschungszentrum für Künstliche Intelligenz
DFKI GmbH
German Research Center for Artificial Intelligence

Founded in 1988, DFKI today is one of the largest nonprofit contract research institutes in the field of innovative software technology based on Artificial Intelligence (AI) methods. DFKI is focusing on the complete cycle of innovation — from world-class basic research and technology development through leading-edge demonstrators and prototypes to product functions and commercialization.

Based in Kaiserslautern and Saarbrücken, the German Research Center for Artificial Intelligence ranks among the important "Centers of Excellence" worldwide.

An important element of DFKI's mission is to move innovations as quickly as possible from the lab into the marketplace. Only by maintaining research projects at the forefront of science can DFKI have the strength to meet its technology transfer goals.

DFKI has 134 full-time employees, including 111 research scientists with advanced degrees. There are also around 130 part-time research assistants.

Revenues of DFKI were about 28 million DM in 1998, originating from government contract work and from commercial clients. The annual increase in contracts from commercial clients was greater than 37% during the last three years.

At DFKI, all work is organized in the form of clearly focused research or development projects with planned deliverables, various milestones, and a duration from several months up to three years.

DFKI benefits from interaction with the faculty of the Universities of Saarbrücken and Kaiserslautern and in turn provides opportunities for research and Ph.D. thesis supervision to students from these universities, which have an outstanding reputation in Computer Science.

The key directors of DFKI are Prof. Wolfgang Wahlster (CEO) and Dr. Walter Olthoff (CFO).

DFKI's five research departments are directed by internationally recognized research scientists:

- Information Management and Document Analysis (Director: Prof. A. Dengel)
- Intelligent Visualization and Simulation Systems (Director: Prof. H. Hagen)
- Deduction and Multiagent Systems (Director: Prof. J. Siekmann)
- Language Technology (Director: Prof. H. Uszkoreit)
- Intelligent User Interfaces (Director: Prof. W. Wahlster)

In this series, DFKI publishes research reports, technical memos, documents (eg. workshop proceedings), and final project reports. The aim is to make new results, ideas, and software available as quickly as possible.

Prof. Wolfgang Wahlster
Director
The MultiHttpServer
A Parallel Pull Engine

Christoph Endres
email: Christoph.Endres@dfki.de

DFKI-TM-99-04
This work has been supported by a grant from The Federal Ministry of Education, Science, Research, and Technology (FKZ ITW-9703).
The MultiHttpServer
A Parallel Pull Engine

Christoph Endres
email: Christoph.Endres@dfki.de

April 1999

Abstract

Internet agents require fast, concurrent access to many web pages. This service should be stable, central, and easy to access independently from the actual implementation language. This paper describes the MultiHttpServer (MHS), a parallel pull engine implemented in Java with a TCP/IP interface for communication with other programs. A second TCP/IP interface provides information for administration purposes. A simple config-file allows application-oriented tuning of the MultiHttpServer. An optional JAVA-Servlet can remotely start up the MultiHttpServer on demand.

The focus of this report is on a user oriented description of functionality and usage of the system. Sample clients in several languages are discussed.

1 Motivation

Internet Agents and Netbots usually deal with numerous information sources, e.g. webpages. Downloading a page is a time consuming operation. On the other hand it is desirable to collect all information required as fast as possible to produce an acceptable runtime behaviour.

The idea of parallel pull is to save time by performing download tasks in parallel. It has been successfully used in several applications developed in project PAN (see www.dfki.de/~bauer/PAN/).

This memo is primarily intended as a reference for users of the MultiHttpServer. In order to develop his own applications based on the MultiHttpServer, the user should be either familiar with the implementation of TCP/IP clients or use one of the sample clients presented later on.

The following sections describe the underlying idea, implementation details, the specification of the MultiHttpServer’s communication protocol, and some sample applications.
2 Architecture of the MultiHttpServer

In this section I introduce the architecture of a single MultiHttpServer instance. As discussed later it can be useful—depending on the required capacity—to have several instances of a MultiHttpServer running. At the moment I focus on the single server instance shown in Figure 1. It has two core components, a server module and several service modules. The server accepts requests from clients\(^1\) by using a TCP/IP interface. For every client connection a service module is generated. The service module consists of two parts. A Request Parsing module controls the dialog protocol with the client and a Request Handler module executes the client’s request by accessing the World Wide Web (WWW) in parallel.

3 Stability Problems

Parallel execution of requests can be done in two different ways: multithreading or concurrent execution of processes. Both approaches are limited by the operating system. Former versions of the MultiHttpServer used multithreading only, which lead to stability problems when the maximum number of threads for one process was reached. Therefore it was necessary to provide a mechanism for creating multiple instances of the multithreaded MultiHttpServer and scheduling the queries. Still it is important to keep in mind that no matter how clever the system resources are used they will always be limited.

\(^{1}\)In this paper I do not distinguish between a human user and a client application when using the word client.
4 Scheduling and forking processes

The current version of the *MultiHttpServer* includes a *scheduler* which supervises all running *MultiHttpServer* instances and schedules user queries. Before actually performing his queries the user asks the *scheduler* which instance of the *MultiHttpServer* he should use. The *scheduler* answers by telling the port number of the least busy *MultiHttpServer*. The user then connects to this port and goes ahead with placing his query.

The *scheduler* regularly (e.g. every three seconds) collects information from the *MultiHttpServers* about their status (i.e. number of running threads). If for any reason a server does not answer anymore, the *scheduler* kills it and starts a new one. In general, servers are created and killed depending on how busy the whole parallel pull engine is. The information about how this generating and killing of processes should be done is provided by a config-file which the user should set up to his requirements and system resources as described in Section 7.

The architecture of the cooperation between the *scheduler* and the instances of the *MultiHttpServer* is shown in Figure 2. In order to establish a communication to the *MultiHttpServer*, the user connects to the *scheduler* via TCP/IP. The *scheduler* checks the capacities of all *MultiHttpServer* running, decides which one should handle the user request, returns the port number, and disconnects the user. The user then connects to the specified port and runs the server protocol as described in the following section.

5 The server protocol

In this section I describe the protocol used for communication between the server and the client. It can be used as a short reference for developers. Sample sessions
are shown in the next section.

- **mode:**2  <one/all>

  There are two different modes for the server protocol, *one* and *all*. When starting a session the first thing one should do is specify the protocol mode to be used.

  Mode *one* means that only one of the pages requested is interesting. This is useful if there are several URLs of webpages providing equivalent information, e.g. three different weather forecast information services.

  In the *one*-mode, they are all requested, but the result of one of them is sufficient and information from other sources no longer interesting. As soon as one requested page returns, all other requests will be canceled and all other running threads killed.

  Information already received from other pages will get lost.

  Mode *all* does not kill running threads autonomously. All of the requested pages will be downloaded (unless killed using the *kill*-command). This mode is used if the information on the requested pages is not equivalent.

  If no mode is specified, mode *all* is used by default.

- **get:** <URL>  [<timeout> [+]]

  Request a URL. Timeout (in seconds) and additional parameters (in case the requested URL is a cgi-script) can be specified.

  Examples:
  
  - get: http://www.dfki.de/
    requests the webpage of *DFKI*. The information should be obtained no matter how long it takes.
  
  - get http://www.microsoft.com/ 2
    requests the Microsoft webpage. If it takes more than 2 seconds to download, the information is not interesting anymore.
  
  - get: http://www.info.edu/pplsearch.cgi 30 +
    first=Arthur +
    last=Rimbaud +
    email= +
    country=fr
    (the ’+’ at the end of a line indicates that further parameter specifications follow).
  
  - get: http://www.info.edu/pplsearch.cgi?first=Arthur&last=Rimbaud

  Let us take a closer look at the last two examples. Both request the same page. The first one by specifying the parameters seperately, one in every line. The parameters are concatenated and written on the URL connection. This is the **POST** method of the HTTP protocol. The second example directly codes the parameters in the request string using the **GET** method. Most CGI scripts handle POST and GET methods in the same way if only a few parameters are specified. One of the main differences is that the POST method can handle much longer parameter inputs, e.g. text.

---

2Please not that the colon is part of the command.
A detailed description of those methods can be found in the HTTP specification.\(^3\)

- **authget**: `<URL> <timeout> <login> <password>`
  Get a password protected URL using specified login and password.

- **info**: `<URL>`
  Get available information about a requested URL.

- **show**: `<URL>`
  Show the output of a requested URL (if available).

- **kill**: `<URL>`
  Remove a no longer needed page or cancel downloading it.

- **shortinfo**
  Get a short overview of the status of all requested pages.

- **stack**
  Show the URLs of all requested pages.

- **stacksize**
  Return the amount of requested pages.

- **available**
  Return the amount of already received pages.

- **rest**
  Return the addresses of pages still to be expected, i.e. all pages of the stack besides those reaching timeout or not accessible for any other reason.

- **more**
  Return the maximum amount of pages still to be expected.

- **success**
  Return one URL of a received page. If there is no page available yet the return value is ‘no’.

- **waitsuccess**
  Return one URL of a received page. Wait until a value can be returned.

- **waitsuccess <timeout>**
  Return one URL of a received page. Wait up to `<timeout>` seconds for a return value.

- **status**: `<URL>`
  Show the status of a requested page. Return values are
  
  - connecting
  - connected
  - receiving
  - received
  - timeout
  - Error `<errorcode>`

\(^3\)See www.w3c.org.
• access trend: <URL>
  Show the access trend of a requested page. Return values are
  - increasing
  - constant
  - decreasing

• access rate: <URL>
  Show the access rate of a requested page in bytes per second.

• poud: <URL>
  Percentage of unfetched data of a requested page. If the size of a page is
  unknown the return value is set to -1.

• size: <URL>
  Show the size of a requested page in bytes.

• help [ <command> ]
  help shows a general help including a list of all available commands, help
  <command> explains the usage of <command>.

• version
  Display version and copyright information.

• bye
  Terminate session and close TCP/IP connection.

6 A sample session

This section shows an example of a MultiHttpServer session. After obtaining a
TCP/IP port number from the scheduler, the client connects to a MultiHttpServer
instance. At the beginning of a session, the server displays a prompt.

+---------------------------------------------------------+
| MultiHttpServer version 1.0        april 99      |
| Christoph Endres, DFKI GmbH      Christoph.Endres@dfki.de |
+---------------------------------------------------------+
Type 'help' for more information
-Ok-

The client now requests three webpages.

get: http://www.dfki.de/ 
-Ok-

get: http://www.microsoft.com/1
-Ok-

get: http://www.whitehouse.gov/10
-Ok-

Using the shortinfo-command, the client checks the current status of the pages he
requested.

\[4\]The size of a document is an optional header field in version 1.0 of HTTP.
The `success`-command is now used by the client in order to obtain the URL of one of the successfully received pages. The page is displayed using the `show`-command.

success
http://www.dfki.de/
-Ok-
show: http://www.dfki.de/

After obtaining the page, the client removes it from the server and then closes the connection.

kill: http://www.dfki.de/
-Ok-
bye
bye
-Ok-

7 Configuration

The `MultiHttpServer` is configured by editing a config file. Here is an example:

```
#
# Configfile for the MultiHttpServer
#

# TCP/IP ports for administrator and user
# If no values are specified, the ports 2000 and 2001 are used by default.
Admin: 2001
User: 2000
```

# minimum and maximum instances of MultiHttpServer running in parallel
# If unspecified, a minimum of 1 and a maximum of 10 is assumed.
Instancemin: 2
Instancemax: 5

# set critical value for the load of a MultiHttpServer instance.
# Unit is pages per server.
# Default value is 50
Load: 25 pps

The values to be specified so far are port numbers for administrator and user, a minimal and maximal number of MultiHttpServer instances running concurrently, and a critical value for the load of a server. The latter is used as an indicator for creating new instances of MultiHttpServer at runtime.

8 Using the administrator port

By connecting to the administrator port some useful information about the status of the system is provided, e.g. the number of MultiHttpServer-instances currently running and the number of their threads.
Example:

MultiHttpServer 1.0
(c) DFKI GmbH 1999
running on serv-200.
You are connecting from serv-200.
Adminport is 2001, userport is 2000.
From 2 up to 5 MHS instances can be created.
Critical load is 5.

2 instances running:
34937: 3 connections, 12 pages.
34956: 5 connections, 17 pages.

9 Start on demand

Using a servlet it is very easy to start up the MultiHttpServer on demand. A JAVA-Servlet enabled Webserver is necessary. An application wanting to use the MultiHttpServer simply creates a http-connection to the startMHS servlet, for instance to:
http://www.myserver.edu/startMHS?command=start.
The servlet starts the MultiHttpServer (in case it was not running yet) and returns the number of the client port and the administrator port (which usually is not necessary since one usually knows which ports one specified in the config file).
10 Clients

Usually the MultiHttpServer is used by other programs, e.g. agents, and not directly by a human user. In this section I provide sample code of how to write a client class in different languages and demonstrate its usage by a short example application. Clients in other languages are under construction.

10.1 An Eclipse Prolog client

The following module specifies an interface to the MultiHttpServer. It can be used for the development of clients.

%%% MultiHttpClient module
%%% Wilken Schuetz, April 1999
%%%

% define module and export public functions

:- module(multihttpclient).

:- export open_connection/3,
   set_mode/2,
   get/2,
   get/3,
   wait_success/2,
   show/3,
   close_connection/1.

:- begin_module(multihttpclient).

% connect to Scheduler, obtain session port and open connection

open_connection(Host,UserPort,SessionStream):-
   socket(internet,stream,UserStream),
   connect(Stream1,Host/UserPort),
   read(UserStream,SessionPort),
   socket(internet,stream,SessionStream),
   connect(SessionStream,Host/SessionPort),
   skip_prompt(SessionStream).

% skip the prompt

skip_prompt(Service):-
   repeat,
   read_string(Service,"\n",_,String),
   String="-Ok-",!.

% set session mode. unfortunately, 'mode' is a reserved keyword in eclipse,
% so 'set_mode' is used instead.
set_mode(Service, Mode):-
    concat_string(["mode ": Mode,"\n"],Command),
    write(Service, Command),
    flush(Service),!,
    read_string(Service,"\n",-"Ok-").

% request a page
get(Service,Page):-
    concat_string(["get ": Page,"\n"],Command),
    write(Service, Command),
    flush(Service),!,
    read_string(Service,"\n",-"Ok-").

% request a page with specified timeout
get(Service,Page,Timeout):-
    concat_string(["get ": Page," ",Timeout,"\n"],Command),
    write(Service, Command),
    flush(Service),!,
    read_string(Service,"\n",-"Ok-").

% wait til one page was successfully obtained
wait_success(Service,URL):-
    write(Service, "waitsuccess\n"),
    flush(Service),
    read_string(Service,"\n",-,URL),!,
    read_string(Service,"\n",-"Ok-").

% show page
show(Service,URL,Content):-
    concat_string(["show ": URL,"\n"],Command),
    write(Service, Command),
    flush(Service),!,
    collect(Service,Content).

collect(Service, Content) :-
    read_string(Service,"\n",-,Line),
    (Line = "-Ok-" ->
        Content=""
    ; (Line = "-Error-" ->
        (!,fail)
    )
    )
    (collect(Service,Rest),
     concat_strings(Line,Rest,Content)
    )
).

.
The usage of the client module is clarified in the following application example. After opening a connection, the client requests several pages, waits for the first page to be received successfully, reads this page, and closes the connection.

:- [multihttplib].
:- use_module(multihttplib).

test(URL,Content):-
    open_connection('serv-200',2000,Service),
    set_mode(Service,"one"),
    get(Service,"http://www.dfki.de/")
    get(Service,"http://www.microsoft.com/",3),
    get(Service,"http://www.whitehouse.gov/",10),
    wait_success(Service,URL),
    show(Service,URL,Content),
    close_connection(Service).

10.2 A Java Client

In this section I present a JAVA client. Please note that although the JAVA SecurityManager does usually not allow applets to arbitrarily connect to sites on the web, it is possible to obtain pages from a MultiHttpServer running on server side. The following class specifies the client:

import java.io.*;
import java.net.*;

// The class Text is basically a Vector of Strings with some special features.
// It is contained in the MultiHttpServer API.
import endres.util.Text;

public class MultiHttpServer {
    Socket s;
    BufferedReader in;
    PrintWriter out;

    // class constructor
    public MultiHttpServer(String host, int port) {
        establishConnection(host,port);
    }
}
void establishConnection(String host, int port) {
    try {
        Socket sock = new Socket(host, port);
        BufferedReader input;
        input = new BufferedReader(new InputStreamReader(sock.getInputStream()));
        String number = input.readLine();
        int sessionport = Integer.parseInt(number);
        this.s = new Socket(host, sessionport);
        this.in = new BufferedReader(new InputStreamReader(s.getInputStream()));
        this.out = new PrintWriter(s.getOutputStream());
    } catch (IOException e) { }
}

public void write(String msg) {
    out.println(msg);
    out.flush();
}

public String read() {
    String line = "";
    try {
        while(line.length()<1) line = in.readLine();
    } catch (IOException e) { }
    return line;
}

public void catchOk() {
    read();
}

// skip prompt at session start

public void skipPrompt() {
    String line;
    while(!(line=read()).equals("-Ok-")) {}  
}

// main protocol starts here: get, mode, kill etc.

public void get(String url) {
    write("get: "+url);
    catchOk();
}

public void get(String url, int timeout) {
    write("get: "+url+" "+timeout);
}
catchOk();
}

public void modeall() {
    write("mode: all");
    catchOk();
}

public void modeone() {
    write("mode: one");
    catchOk();
}

public void kill(String url) {
    write("kill: +url");
    catchOk();
}

public String waitsuccess() {
    write("waitsuccess");
    String ret = read();
    catchOk();
    return ret;
}

public void bye() {
    write("bye");
    read();
    catchOk();
}

public Text show(String url) {
    Text ret = new Text();
    String line;
    write("show: +url");
    while(!(line=read()).equals("Ok-")) ret.addLine(line);
    return ret;
}
}

The following application uses the above client for requesting three pages and displaying the first one successfully received. After finishing this task, the time elapsed is displayed in milliseconds.

import java.util.Date;
import endres.util.Text;

public class examplesession {

    public static void main(String args[]) {
        new examplesession();
    }
}
public examplesession() {
    Date d = new Date();
    long start = d.getTime();
    MultiHttpClient mhc = new MultiHttpClient("serv-200",2000);
    mhc.skipPrompt();
    mhc.modeone();
    mhc.get("http://www.dfki.de/");
    mhc.get("http://www.microsoft.com/",1);
    mhc.get("http://www.whitehouse.gov",10);
    String page = mhc.waitsuccess();
    System.out.println("show");
    Text t = mhc.show(page);
    t.show(System.out);
    mhc.kill(page);
    mhc.bye();
    System.out.println("Time elapsed: "+(new Date().getTime()-start)+" ms.");
}

10.3 A Perl 5 client

Here is a client for the MultiHttpServer written in the popular script language Perl 5. As in the previous sections, not the full functionality of the protocol is implemented but can be added easily, since the interface functions do not differ much.

## MultiHttpClient

### provided by Markus Meyer and Robert Wirth, april 99

package MultiHttpClient;

use Socket;

# constructor

sub new {
    my $class = shift;
    my $host = shift;
    my $userport = shift;
    my $self = {};
    $self->{host} = $host;
    $self->{connected} = 0;
    # request on userport
    $iaddr = inet_aton($self->{host}) or die "no host: $self->{host}";
    $paddr = sockaddr_in($userport,$iaddr);
    $proto = getprotobyname("tcp");
    socket(PAGESOCK, PF_INET, SOCK_STREAM, $proto) or die "firstconn: ";
    connect(PAGESOCK, $paddr) or die "connect: ";
    select (PAGESOCK); $| = 1;
    select (STDOUT); $| = 1;
    # read from userport
    $line = <PAGESOCK>;}
chop $line;
$self->{'PORT'} = $line;
close(PAGESOCK);
bless($self,$class);
}

# connect for a session

sub connect {
    my $self = shift;
    $iaddr = inet_aton($self->{host}) or die "no host: $self->{host}";
    $paddr = sockaddr_in($self->{PORT}, $iaddr);
    $proto = getprotobynice('tcp');
    socket(PAGESOCK, PF_INET, SOCK_STREAM, $proto) or die "mainconn: $!";
    connect(PAGESOCK, $paddr) or die "connect: $!";
    select (PAGESOCK); $| = 1;
    select (STDOUT); $| = 1;
    $self->{connected} = 1;
}

# skip prompt

sub collect {
    my $self = shift;
    my $line;
    $line = <PAGESOCK>;
    if (!defined $line)
        { exit 0; }
    while($line ~ '-Ok-')
        {
            $line = <PAGESOCK>;
            if (!defined $line)
                { exit 0; }
        }
}

# set session mode

sub mode {
    my $self = shift;
    my $command = shift;
    select PAGESOCK; $| = 1;
    print "mode: $command\n";
    $line = <PAGESOCK>;
    chop $line;
    return $line ~ '-Ok-';
}

# request a page

15
sub get {  
  my $self = shift;
  my $okline;
  my $url = shift;
  select (PAGESOCK); $| = 1;
  print "get: ", $url, "\n";
  print STDOUT "status: ", $url, " connecting...\n";
  $okline = <PAGESOCK>;
  if (!defined $okline)
  {
    exit 0;
  }
}

# wait for success  
sub waitsuccess {  
  my $self = shift;
  my $okline;
  my $address;
  select (PAGESOCK); $| = 1;
  print "waitsuccess\n";
  $address = <PAGESOCK>;
  if (!defined $address)
  {
    exit 0;
  }
  $okline = <PAGESOCK>;
  if (!defined $okline)
  {
    exit 0;
  }
  return $address;
}

# show page  
sub show {  
  my $self = shift;
  my $line;
  my $url = shift;
  my $string = "";
  select (PAGESOCK); $| = 1;
  print "show: ", $url, "\n";
  print STDOUT "status: ", $url, " receiving\n";
  $line = <PAGESOCK>;
  if (!defined $line)
  {
    exit 0;
  }
  while($line =~ '-Ok-')
  {

Using the client described above, it is very simple to implement the following small sample session. Please remember that it is necessary to put the location of the Perl interpreter in the first line.

```
#!/project/pan/perl5/bin/perl -w
```
# load module described above
use MultiHttpClient;

# define variables
my $client;
my $address;
my $content;

# session
$client = new MultiHttpClient('serv-200','2000');
$client->connect();
$client->collect();
$client->mode('one');
$client->get('http://www.dfki.de/');
$client->get('http://www.microsoft.com/');
$client->get('http://www.whitehouse.gov/');
$address = $client->waitsuccess();
print STDOUT $address;
$content = $client->show($address);
print STDOUT $content;
$client->kill($address);
$client->bye();

A  Soft- and Hardware requirements

The MultiHttpServer is written in JAVA, i.e. it is platform independent. So far, it was tested under Solaris and Linux, but should run under any operating system providing a JAVA Virtual Machine.

B  Further Information

Further information can be obtained from the MultiHttpServer webpage:

http://www.dfki.de/~bauer/PAN/mhs/

C  Acknowledgements

I like to thank Mathias Bauer for encouraging me to write this paper, Dietmar Dengler and Markus Meyer for interesting discussions on this matter, Wilken Schütz for providing the Prolog client, and Markus Meyer and Robert Wirth for the Perl 5 client.
The MultiHttpServer
A Parallel Pull Engine

Christoph Endres
email: Christoph.Endres@dfki.de