Bachelor-Thesis:
Eiffel HTTP Server

Florian Besser, Scott West, Bertrand Meyer

Chair of Software Engineering, ETH Zurich, Switzerland

Index

Eiffel HTTP Server

Abstract

The HTTP specification is one which is utilized by all web-browsers and servers alike.
More than this, it is a well-known and reviewed specification. This project will deal with the construction of a real-world web-server. The proposed server is implemented in Eiffel, conforms to an agreed upon portion of the HTTP specification (RFC 2616), and uses concurrency to handle multiple connections and facilitate increased throughput.

Summary

The eServer can handle HTTP 1.1 as well as HTTP 1.0 and handles OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE and CONNECT Requests.

Real-world robustness

Our web-server was designed to cope with faulty or even mischievous clients. The problem of keeping connections open indefinitely and therefore pushing the server to it's limits has been addressed, as well as the problem of one particular time-consuming request taking up all of the servers resources. The server has also been tested with different browsers for real-world compatibility.

HTTP compliance

All the headers specified in the HTTP protocol are analyzed and used whenever possible. Requests are searched for request-, entity- and general headers. Responses are sent with response-, entity- and general headers. While reading requests and generating responses, as much information as possible is read in resp. given to the client. This aims for a sound, robust usage even with future clients.

Multi-threaded throughput

Our threads are used to read from and write to sockets, but do so non-blocking. This makes for a smaller memory footprint overall, and less thread switching. The architecture is set up in a way that a reqeust can be parsed line by line, so there is no waiting until the whole request has arrived, or until the whole response has been generated before action can be taken! These features are especially important when for example streaming movies.

Introduction

Since the nineties web-servers became more and more important, as they are the very fundamental „brick“ upon which all Internet services are built. However, this role also means much workload for web-servers, as well as an ever-demanding base of developers to implement more and more features. So web-servers have to be fast, durable and extensible as well as scalable.
The main idea of the eServer project is to build a robust and easily extensible web-server, while also providing good speed and durability.

- Robustness is „inherited“ from the basic Eiffel libraries used, since they are well known to have few errors and run very stable.
- For ease of extensibility, agents are used. The agent mechanism wraps operations into objects. So other developers can define such an operation, which will be executed by the server. A good example of how powerful agents are: Our server is designed to handle HTTP, but with the right agent, it could also handle entirely different protocols!
- Eiffel provides a basic Thread library, which is used to gain more throughput and allows us to make use of multi-core systems. The library works quite well, it is, however, quite slow when switching threads. But if that library will be improved in the future, so will our server.
- Scalability has grown in importance over the last few months: With more and more Denial of Service (DoS) attacks, it is important to have a server that does not break down should multiple users access it at once. For example, if 100 users access a web page at once, the server should be busy for about 100 times as much as if only one user accessed the web page. This is often a problem in real-world web-servers.

So the very fundamental question is: How to build a web-server that satisfies these requirements? The eServer project aims to answer this question, as well as lead the way for an actual implementation. The main contributions of this paper are: (See detailed description)

- Formal specification of eServer behavior / class model
- Architecture explanation
- Implementation explanation (see also code of eServer)
- Guide for extensibility / other guides for developers
- Several guides for real-world deployment of eServers on different operating systems and for different purposes

Goals

Our main goals can be divided into three categories and were set beforehand.

HTTP Compliance

The HTTP Server complies with a good part of the the rules of HTTP 1.1, which means the Server can handle OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE and CONNECT requests, and the answers to these requests are understood and correctly handled by major browsers like Firefox, Opera, Chrome, Safari, IE etc.

An additional goal is that the Headers sent with the requests and answers are also handled according to HTTP 1.1.

Those types of requests the server complies with are supposed to be error-free in general cases and have a low error rate in special cases.

Server Code / Execution

The Server should be multi-threaded, preferably use SCOOP.

The Server should provide a good speed, meaning it is supposed to access more than 100 medium-sized static HTML pages per second.
In a real-world environment the Server is supposed to reach 5% of the throughput-rate of an Apache web-server.

The server is supposed to run stable and smoothly, serving a constant rate of pages per second.

Extensions / Entities

MIME-Support (required to answer many requests, such as GET)

The configuration options of the server allow a flexible deployment and are easily extensible.

Configuration files for:

- Logging
- Error reporting
- Handling of persistent connections (keep-alive etc.)
- Resource control (Number of spawned worker threads, and behavior of the Server under high load)
- MIME types

Related work

There are several other web-servers already operating around the globe, and with sites like Google and YouTube the market value of good web-servers is several million dollars at least. So instead of listing every other web-server, let's have a look at some of the market leaders:

Apache

The Apache HTTP Server Project is an effort to develop and maintain an open-source HTTP server for modern operating systems including UNIX and Windows. The goal of this project is to provide a secure, efficient and extensible server that provides HTTP services in sync with the current HTTP standards.
Apache httpd has been the most popular web server on the Internet since April 1996, and celebrated its 15th birthday as a project this February.
The Apache HTTP Server ("httpd") is a project of The Apache Software Foundation.

LightHTTPd

Lighttpd powers several popular Web 2.0 sites like YouTube, wikipedia and meebo. Its high speed Io-infrastructure allows them to scale several times better with the same hardware than with alternative web-servers.
This fast web server and its development team create a web-server with the needs of the future web in mind:

- Faster FastCGI
- COMET meets mod_mailbox
- Async IO

Its event-driven architecture is optimized for a large number of parallel connections (keep-alive) which is important for high performance AJAX applications.

IIS

Internet Information Services (IIS) for Windows® Server is a flexible, secure and easy-to-manage Web server for hosting anything on the Web. From media streaming to web application hosting, IIS’s scalable and open architecture is ready to handle the most demanding tasks.

Or so they claim. With a market share of about 30% they are obviously doing something right.

Theory

The eServer uses a few but quite important concepts.

Data Transfer over Network

This might seem as a triviality, but there are a few points which make it noteworthy. Data transferred over any network can get lost or delayed, so it is important to deal with these possibilities. Since the underlying TCP Protocol gives some guarantees (for example that data will always be available in the order it was sent), there are certain aspects TCP does not (and probably can't) cover. The two main problems left unaddressed by TCP are:

- No guarantee about round trip time (RTT), so it is possible that a connection times out
- Dignified exit is not always possible. So every packet the server or client tries to send could lead to an error because the connection has already been closed.

If you are interested about how these problems are actually solved, please consider the more in-depth explanation of the server code.

Threads

As already stated in the goals section, Threads are used in this project. Again, this fact alone is nothing new, but there have been some interesting problems on the implementation side, since the Eiffel base library is not thread safe. On the theory aspect, however there is not much news here. If you are interested about how the threads actually work together, please read the detailed description about server behavior.

Detailed Description

This section will deliver a detailed explanation of the key aspects within eServer. If you are interested in the inner working, then go ahead and read it. For most developers and users it will be more informative to read the guides, however. These key aspects normally need not to concern developers and users, but just in case you want to have the same level of knowledge as the inventors, here you go:

Architecture

The architecture of our project is quite ordinary. There isn't any new pattern or some special way how the work-flow is handled. Still, let me give you an overview over the inheritance first:

- APPLICATION inherits from THREAD_CONTROL
- WORKER and CACHER inherit from THREAD
- REQUEST and RESPONSE both inherit from GENERAL_HEADERS and ENTITY_HEADERS, who in turn inherit from HEADER_UTILITIES.
- Additionally, MT_LINKED_LIST(G) inherits from LINKED_LIST(G).

For completeness, let me also document all the patterns and ideas used.
The very first idea is as old as it is simple. Since we had a configuration that was supposed to be globally available, we passed the *configuration* reference to nearly every other object, therefore creating something like **global variables**. Eiffel connoisseurs will protest, that this is not really a neat way, and they are right. But it is fast and comprehensible, as well as easy to access for developers. In the future, someone might code a configuration the same way EiffelStudio uses it: You inherit from a specific class and *configurations* becomes magically available!
Next is a **Producer/Consumer** model for incoming connections. Class APPLICATION listens for them, and inserts a corresponding object in MT_LINKED_LIST, which is depleted by all the WORKER threads. The same process is used again with the agents in MIME, called by WORKER. After a response is generated, they may decide to cache it, so they will supply it to a LINKED_LIST for the CACHER thread (making them Producers in that case). The CACHER thread is the only consumer of that list, and will take the elements given and put them into a better organized SPLAY_TREE.
(The idea was to create only a small additional load to the WORKERs while not having to abandon a fast and scalable storage system like a SPLAY_TREE.)
As already mentioned, the agents play a central role in our system. Every file extension might have a different agent that handles it, and the calling threads have to dynamically evaluate which agent they should use. Which agent is used for which file extension is defined during the creation of the MIME object. And the creation of all the MIME objects happens in the CONFIG.*make* feature, should you want to look. As of now, there is only one handler, the *standard_handler*, so there will be nothing to see.

Implementation

There are two parts of server implementation, the boot sequence and the actual service.
Before a server can run, it must of course start. For this cause it will create a CONFIG object, with the feature *make*. This feature will try to open all necessary configuration files, read them out and convert them into Eiffel objects. Every line in these configuration files must either be empty (equal to „“), be a comment (start with „#“), contain a known command that hasn't yet been executed or it will be logged as „not understood or doubled“.
By making sure the same command doesn't get executed twice we force soundness of the configuration file. For example, it is impossible to reference two files for the „404 - Not Found“ error. Originally in Apache the later-read command simply overwrote the standing command, which has often led to confusion of apache users. If you define your own error file you either want it to work correctly or receive an error – but Apache simply omitted the command.
Any errors logged during this stage will be outputted to *stdout*, so you should always see them, even when starting the eServer as a daemon.
After no more configuration files are to be read, the eServer checks for completeness of the configuration files. Only when all the basic commands have been set will it execute correctly.
There is an alternative mode that adheres to the Eiffel principle of being able to start without any commands at all. If even the basic configuration file is not readable, the server will start with default values (ex: port 1234).

So in conclusion: Either a sound and complete configuration, or none at all. Everything in the middle is prohibited to avoid confusion.

Now that the server is set up to receive connections, the main application will start the WORKER threads as well as the CACHER thread. All threads share the generated instance of the class CONFIG.
All WORKER threads as well as the main application share an instance of MT_LINKED_LIST. We will come back to that object later.
The WORKER threads will block until MT_LINKED_LIST.*count* > 0, and the CACHER thread will wait until a WORKER advises it to cache a resource.
But back to APPLICATION:
The feature *execute* will now loop endlessly while looking for incoming connections. The call to *server_socket.accept* is blocking, so no resources are wasted.
Eventually, a client will establish a connection in hopes of sending a request. As soon as the TCP handshake is completed the connection handler (identified by the client socket) is used to create a REQUEST / RESPONSE tuple, which is stored in the MT_LINKED_LIST. The APPLICATION will also send a signal to all blocked WORKERs, so that they may immediately start working. The APPLICATION has now finished a cycle and will start listening for connections again.

Among the WORKER threads there is now a race who will get the inserted object. As guaranteed by the Eiffel Thread library at least and at most one thread will succeed in obtaining said object. All other threads will look for another object stored (again, with a race) or go back to sleep until they are awoken again by APPLICATION.
The successful WORKER(s) will now try to work their way through reading out the *client_socket* as well as preparing a very basic RESPONSE object.
Let's look at this very important step in a bit more detail. As a first step, we must read out the *client_socket* and fully populate the REQUEST object.
In a request, the first line is very special. It contains the three most important informations of a request. Namely the method (GET, POST, etc.), the resource (/index.html, /Subfolder/xyz.txt) and the protocol (HTTP/1.1, HTTP/1.0). Many requests could be answered just with this line, but there are a bunch of headers that can influence the outcome of the request.
On every line after the first there is exactly one header. An empty line signals the end of all headers. As of now, headers are either recognized and stored in the REQUEST object, or are reported as faulty and ignored. At a later stage, the content of these headers will be evaluated.
There is one exception to this rule, though. Should the header
Content-Length be bigger than 0, then the server knows that the client would like to send an entity with the REQUEST. Exactly as many bytes as specified by *Content-Length* will be read in and stored. Just as before, the content of the entity is not analyzed.
Once the REQUEST object has been completed, the CACHER thread will be asked whether it has an answer to this request or not. The exact behavior of the CACHER is documented in the Behavior section just below this one. Should a RESPONSE be found, the WORKER will immediately deliver it to the client, purge the REQUEST object, and insert the REQUEST / RESPONSE tuple into MT_LINKED_LIST again. It has then completed a cycle and will start listening for new entries in MT_LINKED_LIST itself.
In the unlucky event that not all of the client's data has been received (determined by read errors), the connection will simply be stored in the MT_LINKED_LIST again (but at the very end!). By storing every new entry at the end we ensure a round-robin model amongst connections. A client that opens a connection but doesn't send any data will not impact other concurrent REQUESTs. And until the connection is given to a WORKER again, the missing data will hopefully have arrived. Otherwise, repeat the process until the data arrives or a timeout occurs (default: 3 seconds).
Should the read operation complete without an accurate RESPONSE from CACHER however, then the difficult part starts.
By looking at the file extension submitted in the resource, the WORKER will determine the correct MIME agent, which it will call. Should no MIME type be found, then the WORKER will call the *standard_handler.*
The agent now has the daunting task of turning the REQUEST into a RESPONSE, and to deliver that response back to the client.

To be able to do this, it must check the contents of the REQUEST object, and handle to the behavior specified below.
Once the RESPONSE object has been properly populated with headers, it is time to send the results to the client. Again, there are two possibilities here. If the referenced file is small enough, it will be read in as a RESPONSE entity, and then the whole RESPONSE is sent to the client using the redefined RESPONSE.*out* feature.
In most cases however, the server should not cache the entire file. Just imagine how quickly our server would die when it would start caching 100MB movies. In that event, the RESPONSE.*out* feature is used without defining an entity, therefore sending all the headers to the client, but not yet the entity. The server will then open the referenced file, and transmit it in blocks, effectively streaming it to the client. With this design decision the eServer can wholly cache small requests without having to check back to disc as well as stream big movies with a very low RAM footprint!
After the answer has been sent to the client, there is just two things to take care of: First is the generated RESPONSE, should it be cached or thrown away?
In case the handler decides to cache, it must set all the necessary headers for caching (was done beforehand), and then insert the TUPLE of REQUEST and RESPONSE objects into a LINKED_LIST owned by the CACHER thread. The CACHER thread will later pick up on that TUPLE and will store it in a better ordered and faster accessible SPLAY_TREE.
Once the question about caching has been answered, the agent exits and now the WORKER thread is in control again. The WORKER thread now has to decide the second question: Whether to close the connection or reuse it. If the connection is reused, it will be inserted at the end of the MT_LINKED_LIST mentioned above. (Since there is no input expected on that connection for the next second or so. Usually the client has to make up his mind what to request next, which takes time.)
Otherwise the connection is closed and not used any further.
After that decision the WORKER thread has completed it's cycle and will start listening for entries in MT_LINKED_LIST again...

Behavior

As said above, the MIME agent as well as the CACHER thread behaves according to these rules when converting a REQUEST object into a RESPONSE object. This aspect is technically easy, but it decides whether our server delivers the correct response or just some gibberish. I would compare it to the heart muscle. Not insanely complicated, but still very important.

Let's go through the MIME agent first, since in order to get something from CACHER you need at least one cached REQUEST / RESPONSE pair. All rules that are listed here can be found in MIME.*standard_handler*, should need be.
A word on error handling: Should any operation within the agent fail, it will not send any more data to the client and immediately exit. In fact, every write operation can throw an error. Just keep this in mind, since I won't mention it on every write action.
Now for the rules in chronological order:

- Test if Protocol isn't set or isn't HTTP <= version 1.1
  - Set 505 HTTP version not supported error
- Test for Void method or resource or bad resource
  - Set 400 Bad Syntax error
- Test for method being OPTIONS
  - Set 204 No Content in all cases (different Log entries)
- Test for method being GET, POST or HEAD
  - Set 403 Forbidden if accessing directory or non-readable file
  - Set 404 File Not Found if file does not exist
  - Set 200 OK for GET or POST, 204 No Content for HEAD otherwise
- Test for method being PUT, DELETE, TRACE, CONNECT
  - Set 501 Not Implemented error
  - (CONNECT is reserved by HTTP/1.1 and not used anyway, PUT and DELETE are actually implemented but are too dangerous to be actually called!)
- Test for method NOT being OPTIONS and resource being *
  - Set 403 Forbidden error
- Test for resource being too long
  - Set 414 Request-URI Too Long error
- Test for method being POST without Content-Length header
  - Set 411 Length Required error
- Test for method being POST with Content-Length header being too big
  - Set 413 Request-Entity Too Large error
- Check for Accept-* headers being satisfiable
  - Set 406 Not Acceptable error
  - *Note*: Should the MIME type not specify a language for example, the test for Accept-Language will always pass.

- Test if If-Modified-Since header evaluates to True
  - Set 304 Not Modified
- Check the other If-* headers, if a condition fails
  - Set 412 Precondition Failed error
  - *Exception*: If If-None-Match fails while the method is GET or HEAD, set 304 Not Modified
- Test for Content-Encoding on entity
  - Set 415 Unsupported Media Type error if Content-Encoding is unknown
- Check if Range header isn't satisfiable
  - Set 416 Requested Range Not Satisfiable error
- Test for Expect header
  - Set 417 Expectation Failed error in any case, we don't support this header yet.

Good. Now we have determined the correct response code. If the response code is 2xx, set the required RESPONSE headers. These headers are only used for caching (explained further down) or for the client itself. The behavior of the eServer isn't influenced in any other way by them! (So I won't list them here, it would be a straight copy&paste from the code file. The header „*vary*“ are also the one used for caching, should you have a special interest in that area.)
If the response code is however different from 2xx, the error-file is set using the CONFIG reference. That is also why we only have to set the error codes, and not the error messages themselves.
Now follows the actual response to the client as well as the caching. But this is explained in the implementation section above this section.

Now for the second important part of behavior: The CACHER.
Let me state some general facts about caching in HTTP/1.1 first. You might have come across several lines saying that the REQUEST / RESPONSE pair is cached. Should the RESPONSE alone not be enough? No. The caching process is quite something to behold, since it allows a server to cache even dynamic files.
This behavior is best explained by starting with the data Structure:

```
SPLAY_TREE[
        STRING ,
                LINKED_LIST[
                        TUPLE[
                                REQUEST,RESPONSE
                        ]
        ]
]
```

Yep, and that's all necessary!

Let's look at STRING first. That STRING is like a key. You insert a key into search and you will get the associated LINKED_LIST back. You may have guessed it: That key is the request resource! So if you request index.html, you will get a whole list of possible answers from the cache. Good news is though, that not all of these list items qualify as RESPONSEs. To determine if a RESPONSE can be sent, you must check three things:

- Does the client want cached RESPONSEs at all?
  - Note: This is checked in the WORKER class by looking at the REQUEST.*cache_control* header.
- Is the RESPONSE not expired? Do this by looking at the RESPONSE.*expires* header.
- Is this particular RESPONSE feasible? The RESPONSE will have a vary header field, which is itself a list of REQUEST header fields. Now check both REQUESTs (the cached one and the one fresh from the client) if **ALL** those fields in the vary header are exactly the same. Should this evaluate to **True**, give back the RESPONSE, otherwise keep searching through the list.

Should no feasible RESPONSE be cached, then the WORKER must continue to evaluate the REQUEST. (See handling above)

Developers Guide

Hello, future developer!
First, please consider the normal installation procedure as described in the users guide. Once you have your eServer set up and the code loaded in EiffelStudio, we shall begin with this short tutorial.

Let me give you a short overview where the most important parts of the eServer are:

- Reading Config: CONFIG class, feature **make** called by APPLICATION
- Starting and observing Threads: APPLICATION
  - Threads include both WORKER and CACHER threads.
- Accepting / Storing new connections: APPLICATION (See feature **receive** in APPLICATION and Feature **insert** in MT_LINKED_LIST)
- Storage for these connections is MT_LINKED_LIST
- WORKER is responsible for depleting that storage. (Feature ***item_and_remove*** in MT_LINKED_LIST and **receive** in WORKER*)*
- Should the connection already have timed out the WORKER will drop it now. (See feature **timed_out**.) For the purpose of reading from a connection, WORKER stores all info in an object of type REQUEST. (Feature ***read_message*** in WORKER, call to different features in REQUEST)
- The WORKER then submits this REQUEST object to the CACHER Thread, which might have a cached resource already available. (Feature **search** in CACHER)
- If there was no response available in the cache:
  - An agent, defined in MIME is then called with this REQUEST object as well as a RESPONSE object as arguments. This agent will generate the appropriate RESPONSE object, as well as giving the final response to the client. (For example see ***standard_handler***)
  - Which agent is called depends upon the file extension. To specify which agent should be mapped to which file extension, see the **make** feature of MIME. (Later we will extend our system by more agents.)
  - If the agent deems his response to be cache-able, it will submit it to the CACHER thread, who will then store it and wait for future requests. (Feature **insert**)
  - After WORKER has called the agent, it will either reuse the current connection by storing it in MT_LINKED_LIST, or close it. It will then wait for the next connection. (See feature **close**)
- Otherwise, the found response is delivered to the client unaltered, directly by WORKER.
  - After WORKER has sent the response, it will either reuse the current connection by storing it in MT_LINKED_LIST, or close it. It will then wait for the next connection. (See feature **close**)

After the more theoretical introduction, let me give you some concrete examples for extending and altering the eServer:

Adding new features with options in the configuration files

- The time might come when the current configuration files are not good enough anymore, and need to be extended. There is a general rule here: Whatever needs coding or appeals to a professional user base goes into code files. Whatever needs to be changeable by users without Eiffel compilers goes into text files.
- As an example, we will try to add virtual server support to eServer. This is a feature that will of course require some code changes, but it is essential that also normal users can access it via configuration files. This example will just show the configuration part, and not the actual implementation.
- First, since this is a new feature, create a new configuration file. I have already done this, the file is called hosts.conf, you should find it in **/var/eServer/Config Files/**.
- The syntax this file is written in is similar to the one used by apache, so that users should feel familiar. You may keep up this practice or abandon it, that's your choice.
- Now that the file is in place, it needs to be loaded. You can load additional configuration files by adding the following line to **/var/eServer/Config Files/ehttp.conf:**
  Include /var/eServer/Config Files/hosts.conf
  (Change the path so that it fits the newly created file!)
- You are now set on the non-Eiffel side. Now come the code changes. Open EiffelStudio and navigate to the CONFIG class.
- If your changes (as in this example) require information to be stored for further use, create new fields in this class. (Here we use the new field **virtual_host**: *LINKED_LIST[STRING]*.) A CONFIG object can be referenced from nearly anywhere in the code-base, so do not worry about how to get that info to the right spot later on.
- Once you have defined the necessary fields, you need to change the **make** feature to actually read in the info. I apologize for the spaghetti code, but this portion of the eServer is hardly the most important. Look for the line
  If config_string.as_lower.starts_with ("serverroot")
  There will be many **elseif** clauses below that line, and you must add your own to this list. Example:
- ```
  elseif config_string.as_lower.starts_with ("<virtualhost") then
      log_message ("New VH!")
      --Add VirtualHost to some sort of list
  elseif ...
  ```
- Good! Now you can reference a list of virtual hosts from nearly any part of the code-base by writing **config.virtual_hosts**.
  ```

Adding a new handler

- For example for .php files:
- Open MIME.e
- Add the following feature to MIME.e, possibly in the „internal“ feature group:
- **php_handler (req: REQUEST; res: RESPONSE)**

```
do
        --System calls go here
        --Something like:
        --system ("/etc/php5/php -f "
        config.base_path + req.resource)
end
```

- Change the feature **make***:*
- **make (e: STRING; l: CONFIG)**

```
do
        extension := e
        config := l
        if e.is_equal ("php") then
                set_handler (agent php_handler)
        else
                set_handler (agent standard_handler)
        end
end
```

- You are good to go. Change the **php_handler** to your liking.
- *Note*: The system call shown in the comment above will actually work, but beware that headers and client information etc. will not be given to the PHP executable! So for example backup scripts work fine, but a script that should mirror back the user's IP address won't!

Adding a new header

- If you would like to use more headers than defined in HTTP/1.1, you can of course add these. For this example we use the Header **xyz**
- For Request / General / Entity Headers, open REQUEST.e, GENERAL_HEADERS.e or ENTITY_HEADERS.e respectively.
  - Add the field **xyz: TYPE_OF_XYZ** to the class
  - Add the features **get_xyz**: STRING and
    set_xyz (**x: TYPE_OF_XYZ**) to the class
  - Change the feature **get_header_** in REQUEST.e so that if handles the new header **xyz**
- For Response Headers, do the first two steps and then **don't** change **get_header_**, but change the feature **out** in RESPONSE.e, so that it handles **xyz** just as any other header.
- You have now defined a new Header. However, it does not yet „do“ anything (it is read in and printed out, though!).

- You must now change the handler for the appropriate files in MIME.e. Say you would like to extend the **php_handler** which we have added above. You could now write:
- **If req.xyz.is_equal (*some_reference_object*) Then**
  --Code your changes her
  end
- If your new header should be supported by all extensions, you should change **standard_handler**, also in MIME.e
- If you would like to check your new header field first, and then decide which other handler to call, you can do that, too! Simply set up a „dummy“ handler for the appropriate file ending, which then calls any other handler with the arguments it received itself.

Adding new error codes

- As already mentioned in HTTP/1.1, there could be more error messages or general status codes in the future. It is quite easy to extend the eServer for this purpose. (For this example we use the error code 123)
- *Warning*: Only Error codes between 100 and 999 are supported!
- First, let's do the non-Eiffel part. Add the line
  ErrorDocument 123 /path/to/error/files.html
  in file
  /var/eServer/Config Files/ehttp.conf
- Now create/edit the errorfile you just referenced, so that it contains a meaningful HTML message.
- You need only add one line in the Eiffel files. Add the line
  errormessages.enter ("Your Error Summary", 123)
  to the file CONFIG.e (in the feature **make**, together with all the other error definitions.)
- Good. In theory you now have created your own error message. In practice however, this error never happens. So you should add some code to the handlers found in MIME.e. For example:
- **If req.some_header.is_equal ("some value") Then**
  res.set_code (123)
  end
- The rest will happen automatically.

Users Guide

Welcome future user. If you have happened to scroll over the developers guide, you might think you are on hell's doorstep. If you happen to be a student which already had the „pleasure“ with Eiffel for the first time, you might be tempted to just throw your laptop away. But fear not! As long as your EiffelStudio does not quit unexpectedly, everything should go just fine!

Preparations

Make sure you have EiffelStudio installed and that it runs okay. Make sure you use the most recent version!

Where to find and download eServer

As of now, it can be obtained through the SCOOP project svn. I simply assume that you have access and can check out the source code, or this tutorial will turn into a book. Should you be unable to check out the source code, try asking for help. Figuring out this kind of stuff takes ages! Just register an Origo account and read through the tutorials, or ask on some related forums.

How to build the solution

Alright, you have the source code, and it's time to import it into EiffelStudio. Use the „Add Project“ button and navigate to the folder where you checked out the source code. You should look for a eserver.ecf file. There are some other testing utilities coming with the eServer project, just ignore these for now.
Once the project is loaded, try to compile it. If your EiffelStudio is set up correctly this should work, but produce a few warnings. These warnings are nothing to worry about.
Congratulations! You have just built your very first version of eServer. That wasn't so bad, right?
Please nod now, the hard stuff hasn't started yet ;(

Should you be on Windows. Now would be a good change to read the Windows guide! Your program might compile fine, but it won't run without the changes detailed in the windows guide!

Also make sure your port 80 is free, or change the port in the configuration files!

Alright, while the windows users applied the windows guide and the UNIX users boasted about the supremacy of their OS, we can now get to actually *running* the eServer. Just click the „Run“ button in EiffelStudio. If everything was correctly set up, you should get many lines of data to your *stdout*.

This could be called the „boot sequence“ of eServer, it reads in all the configuration files and tries to store the info given within these files. Again, there will be some warnings and maybe „not understood“ messages. Look at those closely. Most of the time, they are harmless, but they can screw up your eServer behavior if left unchecked.

Now finalize the code and you are fully operational.

Windows Guide

The eServer was originally developed for Linux., so there are a few changes necessary before Windows users may use the solution.

Path settings

Since the paths to files are different in Windows, you need to manually set the paths to the correct file. First, open the file CONFIG.e with any standard text editor, and look for the line
create main_config.make ("/var/eServer/Config Files/ehttp.conf")
Now change the path to your *ehttp.conf* file, for example
create main_config.make ("C:\Eiffel\eServer\Config Files\ehttp.conf")
Once you have done that, access your *ehttp.conf* file, it might contain more paths to other config files. Change those paths to match the correct files.
The configuration files referenced by ehttp.conf might have paths to different files in them! Make sure you do not only go through ehttp.conf, but go through all other files referenced!

Correct Read/Write access

This should not be required by default, as you'll probably have enough rights, but who knows in what direction Windows will evolve.
You should probably run as an administrator, since you'll have the least problems when doing so.
Make sure you yourself can access and change all the files specified in the configuration files. If that works, you should be good to go.
In case you are concerned about security, and would like to run the eServer as a normal user: Switch to Linux. I recommend Ubuntu for beginners.

Conclusion

Over the course of this project, many things changed and needed to be re-evaluated. But here is the final conclusion about what worked and what didn't:

Behavior

The correctness of the eServer was always a very important aspect (not too surprising, isn't it?). And as it turns out, we even exceeded our initial goals. Not only can we handle the 8 main requests types, but we also understand the headers that come with them.
The correct behavior was manually verified and the stability of the server was automatically tested (see testing projects).
A good example for expected behavior is our compatibility with Firefox. Once we deliver a file to Firefox, it will remember our headers and upon the next request not ask for the file itself, but rather whether it was updated. So instead of resending the whole file, we can just send a 304 not modified response and be done with it! Firefox will of course not display the error message but instead the cached file to the user.

Architecture

The idea of a threaded architecture worked out pretty well. For examples about how effective the current architecture makes developer changes, please read the developer guide.
The idea of switching threads twice per request initially came without much thought, and has proven to sometimes hamper performance. But unfortunately I don't see a solution for the time being. There are a few ideas like letting each WORKER having it's own cache (eliminating the CACHER thread, but bloating up RAM consumption) or letting the WORKERs search the cache directly, without a thread for CACHER (works fine on small load, but becomes slower when all the threads have to wait for each other).
There is one bigger problem however. Since we need access to the configuration option nearly at all points in the code-base, we had to share such an object. Since there are only readers and no writers that isn't a problem in the current state.
The problem becomes obvious when converting using SCOOP. Please read the summary about SCOOP for more information about this problem, though.

Implementation

At the beginning, implementation seemed pretty straight forward. But then came the bugs and other problems. In the end, the problems could be solved pretty convincing, with only minor penalties to speed, <and no penalties to functionality.
The problems and solutions are described in the detailed explanation, but let me list the actual impacts these workarounds had here:

- Since the base library isn't thread-safe we lost some precious milliseconds because we had to design our own thread-safe LINKED_LIST implementation, called MT_LINKED_LIST.
- Curiously, file-level commands seems to be thread-safe, so both writing to our log-file and reading from files on disk never needed any improvement.
- SOCKET**.read_line** (see detailed explanation) wasn't fixed. This makes for the ugly SOCKET**.read_line_until** (1000000), which is slightly slower.
- SOCKET.put_string isn't working correctly. The expectation was, that the data written to the socket would actually be transmitted to the client. Should the socket be closed however, the data was simply discarded. A call to SOCKET.cleanup should block as long as data is still unsent!
- The solution was to switch to sending PACKETs instead of STRINGs, which is considerably more work. (And also slower!)
- The additional problem of having to convert STRINGs to C_STRINGs for the generation of PACKETs didn't speed up the process either.

SCOOP

SCOOP is kind of a mash-up between architecture and implementation. Since it is also a quite unique technique, I decided it deserved it's own explanation.
I was originally trained on the formal, theoretical explanation of SCOOP, before there was an implementation. In the meantime, an implementation was drafted and is also working fairly well in itself. This created the first and most basic problem when programming with SCOOP. The theoretical approach assumed a SCOOP-safe base library, and the examples mirrored that. However, there were changes to the syntax in the actual implementation, which threw me off course. Once I found my way back to decent levels with trial&error, the next problem unfolded: The base library exactly wasn't SCOOP-safe.

Let me give you an example why SCOOP caused these problems and how: The CONFIG class.
Every thread used to have a reference to a CONFIG object, but since I wanted to use SCOOP, this was of course no longer possible just like that. I had to define *config*: **separate** CONFIG. This alone wasn't a problem, but what about a different processor wanting to change the CONFIG object? Obviously, this possibility must be considered, otherwise we would cut down extensibility.
Consider the simple case of a processor having a STRING, and wanting to attach it to a STRING in *config.*
According to theory, it could just call
config.the_string.*append*(my_string).
The original idea was, that the processor assigned to *config* would now lock my_string, and use it. But since the call was placed on a **separate** object, the argument should also be **separate**. At first I didn't give it much thought, and dutifully changed the type of my_string from STRING to **separate** STRING. Now the SCOOP problem is solved. But we are not finished yet, as *append* is defined by the base library, and can't handle **separate** arguments.
As it turned out, the only way to solve this problem was to lock *config* on the current processor, then converting *config*.the_string from type **separate** STRING back to STRING, then *appending* my_string (which is now done locally, not remotely!) and then setting *config*.the_string with the result.
This defies any speed improvements that SCOOP could have brought with it, and at that point I decided to just leave it rest until a final implementation is delivered by the people at EiffelSoftware.

Evaluation with respect to time and space

The last changes cost us a bit of speed, bot overall the project did very well. On uncached requests we beat the Apache standard installation. For 10'000 requests (all uncached, to files that didn't exist) we needed 7.3 seconds real-time, while Apache needed 9.8 seconds.
However, we used more RAM. This could be some overhead from Eiffel, that will not matter in bigger deployments, or a real problem with the eServer. The 10 MB the eServer used for 10k requests seemed so little though, that further evaluation was not deemed necessary.
For 1 GB of RAM you could cache a million requests, that should be enough, overhead or not ;)

I would specially like to point out our goal at the beginning: Reach 5% throughput of an Apache web-server, and access 100 static HTML pages per second. As of now we reach about 134% throughput of apache, and about 1020 requests per second (uncached). With these figures we definitely exceeded all expectations in this area!

Future Work

We might have laid an important base for other projects, but the eServer is far from the capabilities of an apache web-server, for example.

Extensions

As of now, there are many other programming languages that are favored amongst web developers. Unfortunately, the eServer does not yet understand these languages. Future extensions for languages like PHP are eagerly awaited.

Library performance improvement

I don't like the idea of blaming bad performance on your underlying libraries, so consider this more of a note. Should the performance magically improve, then even the eServer will run faster. Let's just hope that it will ;)

More configuration possibilities

One aspect I really look forward to is the extensibility also on the config files. With our extensible system we have the perfect base to listen to our users and implement their wishes. We neither have to deal with obscure C files directly nor do we have to lower our throughput for additional modifications to be loaded.

References

Hypertext Transfer Protocol – HTTP/1.1 (RFC 2616)
Found at: http://www.w3.org/Protocols/rfc2616/rfc2616.html

Nienaltowski P.: Practical framework for contract-based concurrent object-oriented programming
Found at: http://se.ethz.ch/people/nienaltowski/papers/thesis.pdf