FastCGI — The Forgotten Treasure

Peter Simons

simons@computer.org

Ralph Babel

rbabel@babylon.pfm-mainz.de

Table of Contents

Abstract

Meeting the demands on modern web sites is more difficult than ever: People are no longer interested in "home pages"; they want web-based applications that provide useful functionality. Thus, the complexity of the average web site has increased dramatically for the programmer. Furthermore, the performance requirements have gone up: The faster the Internet gets, the less are people willing to wait. At the same time, though, the number of users has increased by an order of magnitude!

Facing this situation, web developers ought to ask themselves: "Are the tools we use to write web-based applications still adequate?" And there are some that begin to realize that the answer may be "no". Even though tools like PHP are becoming more and more powerful, they limit us with inherent architectural constraints. PHP may be fine for layouting HTML pages and including some dynamic content, but it provides only insufficient mechanisms for the programmer.

This paper shows developers how high-performance web applications can be implemented quite easily using the FastCGI interface and C++. After a brief introduction to the FastCGI interface, a generic C++ framework for web-based applications using FastCGI will be discussed, including the advantages and disadvantages of this approach.

All source codes shown in this paper are available on the word-wide web at [Simons2002].

1. Introduction

Ever since the web has been invented, people wanted to do more than just provide nicely layouted texts with the occasional picture or link thrown in; there has always been a demand for mechanisms that allow for the creation of *dynamic* web pages. For many years, the only way to achieve that goal was the CGI script: an application using the Common Gateway Interface to receive and process HTTP requests from the web server. CGI scripts were typically written in Perl or — less often — in C and implemented things like page counters or guest books.

More sophisticated developers, however, created entire applications using the CGI interface. One fine example for such an application is the bug-tracking system "Jitterbug", available at [Jitterbug]. Jitterbug, like many other CGI-based applications, is not a "web page" in the classical sense. It is an *application* just like StarOffice, Emacs, and many others — only that it does not provide its user interface via X11, but via the web. It is a "web-based application".

A few years ago, such web-based applications were the exception rather than the rule. The whole web technology wasn't sufficiently advanced to allow for complex applications, neither was the Internet's bandwidth or the average web developer's skills. So people went ahead and wrote guest books and page counters to familiarize themselves with HTML, HTTP, and the web as a whole.

Soon, the Internet community realized that CGI had drawbacks: Writing a CGI is somewhat uncomfortable if all you want is to process a few strings and display the results. So some clever people came up with entirely different solutions like SHTML, ASP, and PHP — to name a few. The difference between CGI and say PHP is that CGI is an *interface* between the web server and the application. All CGI does is to specify how an external application can receive HTTP requests from the web server; CGI does in no way tell the developer what that application should look like or what that application should do with the HTTP request.

PHP, on the other hand, is a specialized programming language, usually built directly into the web server, that provides some simple mechanisms to do what web developers want to do: read some data, process them, display the results as part of a dynamically generated web page. If you are trying to implement a page counter or a guest book, PHP is way superior to CGI; in PHP you can do things like that in only a handful of lines of code, while the equivalent CGI script would be at least ten times bigger. (If you think that this statement is not correct, please consider that in a CGI script you have to implement the entire CGI protocol! In PHP, the programmer does not have to worry about, for instance, parsing POST or GET headers.) Thus, it is hardly amazing that PHP and its alikes became very popular.

Nowadays, PHP absolutely dominates the development of dynamic web pages. The number of functions included in the most current PHP release is mindboggling, and so are the web sites people set up using PHP. One of the most popular sites implemented in PHP is the open-source software-development community “SourceForge”, which can be visited at [SourceForge]. Certainly, a web site like this is not in the same league as a page counter or a guest book anymore; this is a web-based *application*, an application providing complex functionality over the web.

The same applies to many other systems available on the Internet today: There are myriads of shop systems, on-line brokers, search engines, databases, appointment calendars, and chat rooms on the web. All of them provide complex functionality, and virtually all of them are written in PHP. Apparently, the demands on web sites have changed: Users are no longer interested in something that just looks nice, they want something useful! So the web-based application is coming after all.

Today’s web developers are no longer artists who layout an HTML page that is nice to look at. They have to be software engineers who write “real” applications. Realizing this, we should re-consider our choice of tools, just as we did back in the good old days when everybody shifted from CGI to PHP.

PHP is just fine for the creation of dynamic web pages. It offers hundreds of ready-to-use functions that manipulate strings in any conceivable way; it allows for the comfortable creation of even complex HTML structures; it supports simple and straight-forward access to database systems; and so on … However, the truth is: PHP, as a *programming language*, sucks. Big time.

PHP features such esoteric functions as `mcal_is_leap_year()` — a function that will tell you whether a given year is a leap year —, but it does not even understand a simple and important mechanism like the “structure”, a datatype the inclusion of which in virtually every programming language invented ever since Pascal reared its ugly head is not a coincidence. Neither does PHP provide type safety. Neither does PHP support the concept of a link library. The list goes on and on. Many concepts that have proven to be immensely valuable to software engineers throughout the last few decades are unavailable to PHP programmers.

These shortcomings weren’t important when PHP was used to implement page counters and guest books, but they sure are important now that PHP programmers write applications adding up to ten thousand lines of code and more. PHP was never meant to be used to write *applications*. Consequently, you shouldn’t.

Programming languages like C and C++ suit the application programmer better than PHP does. So why shouldn’t they suit the web-based application programmer better, too?

This paper is intended to give web developers a new perspective at how web-based applications could be developed. Actually, it is an old perspective, because all the technology we use has been around for more than four years now. The reason why these mechanism haven’t been used much in the past is that there was no need to. The time wasn’t ripe for advanced web applications yet. Nowadays, however, with millions of potential users in the Internet, developing web-based applications is more attractive than ever. Hence, there is no reason *not* to develop complex web-based applications, but why make it more difficult than necessary?

2. The FastCGI Interface

We already mentioned the CGI interface, which defines a way for web servers to delegate HTTP requests to external applications. This is exactly what web-based–application writers need: a way to connect their application to the web. The Common Gateway Interface is very well-suited for that purpose. Actually, though, there is an even better interface readily available: the FastCGI interface.

As the name suggests, the FastCGI interface is merely a variation of the CGI interface we all have known for years. At the protocol level, the two interfaces are identical. Both FastCGI and CGI applications receive the very same information from the web server, only that they receive it by different means of communication. The communication between the web server and the CGI script takes place via environment variables. As soon as an HTTP request is received, the web server creates a new process — the CGI program — and passes a set of variables to that program in its environment. Variables like `$HTTP_USER_AGENT` and `$QUERY_STRING` will be set, telling the CGI script about the client's browser version, values passed via the GET method, etc. Then the web server reads any data the CGI script writes to the standard output channel and passes them verbatim to the client that requested the page.

A FastCGI application does exactly the same, only that it communicates with the web server via interprocess communication (IPC), such as a UNIX socket or a TCP socket. Upon receipt of the HTTP request, the web server connects to the FastCGI application, transfers the information as defined by the Common Gateway Interface standard and reads any data that are coming back in order to pass it on to the client.

So if that's the only difference, what's the big deal? Why would anyone want to use FastCGI over CGI? The difference is subtle but important: Using FastCGI over CGI allows for an entirely different architecture of web-based applications. In CGI, the lifetime of the application is tied to the lifetime of an HTTP request. For every HTTP request received, the CGI script is started in order to process the request. Once the request is processed, the CGI script has to terminate in order to let the web server know that the request is done. This is different for FastCGI applications; a FastCGI application potentially lives forever: Once started, it may process an arbitrary number of HTTP requests without ever exiting. This architecture has numerous advantages, which will be discussed in great detail in Section 4 of this paper. Before we come to that, though, we shall study the protocol itself a bit more in-depth.

2.1. The Communication Protocol

The FastCGI interface, as specified in [FastCGISpec], is a packet-oriented communications protocol. Even though the underlying transport mechanism is connection-oriented, the FastCGI protocol is not. (Yes, there is a very good reason for this. Just read on.) Each message consists of a header and a body. The header contains — among others — the following fields:

- *Version*: Identifies the FastCGI protocol version. The most current version of the protocol as of today is version 1.
- *Type*: Identifies the FastCGI message type — the general function that the message performs.
- *Request ID*: Identifies the FastCGI request to which the packet belongs.
- *Content Length*: The number of bytes in the body of the packet.

Furthermore, the following message types are defined:

- *BEGIN_REQUEST*: This message is sent from the web server to the application in order to begin a new request.
- *ABORT_REQUEST*: This message is sent from the web server to the application in order to abort an already running request. This occurs, for instance, when the user presses the browser's "stop" button while the request is being processed.
- *END_REQUEST*: This packet is sent from the application to the server in order to tell the server that processing an active request has finished. The body of this packet will contain a "return code", which determines whether the request has been dealt with successfully or not.
- *PARAMS*: This is a so-called "stream packet", sent from the web server to the application. "stream packet" means that multiple packets of this type may be sent per request. Each packet contains data in the body of the message, which are to be processed by the application. Upon receipt of an empty packet — one with a content length of

zero —, the stream ends. The PARAMS stream contains exactly the same information as defined in the Common Gateway Interface standard (see [CGISpec] for further details). Rather than setting the variables in the environment, though, the server will send them one at a time via this message type.

- *STDIN*: This is a stream packet, too. On the STDIN stream, the application will receive the POST-method--related information — if there is any. Again, the format of these data follows the CGI standard.
- *STDOUT*: This is a stream packet, too, but this stream flows from the application to the server and contains the data to be sent to the user who requested the page.

The typical conversation between the web server and the FastCGI application looks like this:

- The web server receives a request that is supposed to be handled by the FastCGI application. Thus, it tries to connect to the application process via either UNIX-socket or TCP-socket I/O. (Which one is used depends on the web server's configuration.)
- The FastCGI application sees the incoming connection. It may now choose to accept or to reject the connection. If it accepts the connection, the application will try to read a packet from the new stream.
- If the FastCGI application does not accept the connection within due time, the request fails. Otherwise, the server will send the application a BEGIN_REQUEST packet with a unique request ID. All subsequent packets will have the same ID noted in the header.

  Then, the server may send any number of PARAM messages to the application. Once all variables have been transferred, the server closes the stream by sending an empty PARAM packet. Furthermore, it will forward any POST data it receives from the client to the application via the STDIN stream. Once all data have been forwarded, an empty STDIN packet is sent to close the stream.

- Meanwhile, the application has received the BEGIN_REQUEST packet. It may either reject the request by responding with an END_REQUEST packet, or it may accept and process the request. If the application accepts the request, it will typically wait until all PARAMS and STDIN packets have been received. Then, the application will process the request and write any output to the STDOUT stream. Once processing is finished, the application will close the STDOUT stream by an empty STDOUT packet and finally send the END_REQUEST message to let the web server know whether an error occurred or not.

Some of you may wonder now why the message header contains a request ID. After all, if we have one connection per request, that type of information is redundant. That's right, only that there may be *more than one request* per connection! That is the reason why FastCGI uses a packet-oriented protocol rather than a single data stream: Multiple requests can be intermixed in one connection; software engineers will know that as "multiplexing". Because every packet contains a unique request ID, the server can deliver an arbitrary number of requests simultaneously — using only a single connection. Furthermore, the FastCGI application can receive data for an arbitrary number of connections, each of which can carry an arbitrary number of requests.

Anyway, the description above misleadingly suggests that the communication takes place sequentially. That is not the case: The server may well send say twenty BEGIN_REQUEST packets in a row, then follow up with a few PARAMS packets from each request, then send some STDIN packets, then send some PARAMS packets again, and so on.

To make this whole thing even more flexible, the web server may also maintain multiple connections to one application, while continuing to multiplex multiple requests on each connection. The web server may just as well maintain multiple connections to several distinct FastCGI applications, all of them working in parallel. And since the communication may take place via TCP, these FastCGI applications may also run on different machines.

2.2. The Wonders of Multiplexing

At first glance, this architecture looks like a nightmare to implement, but in fact writing an application capable of multiplexing several requests is pretty easy. We shall study a generic framework for FastCGI applications in Section 3 of this document, which allows the developer to write even complex applications without having to worry about every detail of the FastCGI protocol. But before we come to that, one more word about why anyone might want to use such an architecture at all.

Multiplexing several I/O channels within a single process is a technique that has been somewhat forgotten in recent years — probably because nowadays almost every operating system supports preemptive multitasking. Most developers seem to assume that with preemptive multitasking available, implementing applications capable of internal multiplexing isn't worthwhile anymore. This is wrong. Without a doubt, preemptive multitasking has innumerable benefits, and certainly nobody would ever again want to work on an operating system without it. But, having three separate processes running simultaneously just to read and write from three different I/O channels is a waste of resources.

When we compare an implementation of an I/O-bound application that uses one process per channel to an implementation that is capable of multiplexing the channels internally, we'll find some major advantages of the latter approach:

- An application that multiplexes internally can handle context switches way more efficiently than the operating system. The OS has to re-programm the MMU; it has to restore the contents of CPU registers; it has to re-construct the memory image of the process; etc. All these operations are not necessary when multiple contexts are managed by the application itself.

  Applications capable of multiplexing can *easily* handle several hundred channels simultaneously. Hardly any operating system can cope with that number of active processes, though. The overhead for the context switches would become so large that the machine would waste most of its CPU time for administrative tasks rather than running the actual processes (thrashing).

- An application capable of multiplexing internally needs way less resources for its task; typically, the context of each I/O channel is stored in a data record (or "structure" in C-speak), and one copy of that record is kept per channel. An application that controls only a single I/O channel per process duplicates the *entire process environment* for each channel!
- Applications that use internal multiplexing can schedule their tasks more intelligently than the OS's general-purpose process scheduler can. The developer can control exactly at what points of program execution context switches may occur; the developer relying on preemptive multitasking cannot.

With FastCGI, we have an interface available that allows us to use internal multiplexing between several HTTP requests. That is one reason why well-designed FastCGI applications outperform "normal" web-based applications by an order of magnitude. An application capable of multiplexing can, upon receipt of a new request, send out the database query. While the database engine fetches the requested records, the application can send out a few pages that were requested a bit earlier and have been created in memory already. Then, it checks for results from the database again ... nothing yet. OK, there is a new request coming in anyway, so we process that one first and send out the next query to the database. Meanwhile, the first query has returned results, so we start reading those via IPC. Reading the results may block after a kilobyte, though, because we can read those results faster than the database can come up with them. So we look at the second query again: no results yet. OK, then we start processing the results we already have from the first query. Once that has been accomplished, we try to read from the database again, etc. ...

All this happens within a single process; no context switches by the OS are necessary to achieve that. Of course, multiplexing will not make the CPU run any faster than it did before — even with multiplexing, the machine can handle only so much load, but we can use that available processing power *more efficiently*. Consequently, the response time for the user improves. As soon as his results are available, they will be processed. And while the results are not available, we process somebody else's.

2.3. FastCGI Support in Apache 1.x

Apache 1.x does not come with out-of-the-box FastCGI support, but there is a third-party module available that adds FastCGI support to Apache: mod_fastcgi. The module can be found at [ModFastCGI].

Unfortunately, mod_fastcgi is not capable of multiplexing several requests over one connection; it needs one connection per request. It cannot even re-use that connection after a request has ben processed; mod_fastcgi will connect to the application every time a new request is received.

We hereby express our hope that this will be remedied in the near future. In defense of the mod_fastcgi author, though, it should be noted that Apache *itself* is unable to handle multiple requests within one process, so obviously there is little that the mod_fastcgi author can do about that shortcoming. Luckily, there are — contrary to what some people might believe — *other* web servers available that don't suffer from this limitation.

3. Writing Web-based Applications using FastCGI

Apparently, FastCGI applications are supposed to handle an arbitrary number of concurrent requests per connection, while at the same time handling an arbitrary number of concurrent connections — and all of that in a single process. Sounds scary, doesn't it? How do you write an application like that?

The truth is, if you try to build such an application from the scratch, you will most likely have an headache or two before everything works. Fortunately, you do not have to do that. In this section, we will describe three reusable components, developed exactly to make writing multiplexing applications as easy as possible. Put together, these three libraries provide a framework for FastCGI application programmers that is extremely powerful yet comfortable.

We will discuss these components separately before concluding with an implementation of a simple FastCGI program that makes use of the framework.

3.1. Scheduler Library

The scheduler library contains a single class, declaration of which is shown in Figure 1.

Figure 1. Declaration of the `scheduler` class

```
class scheduler
    {
  public:
    class event_handler
        {
      public:
        event_handler() { }
        virtual ~event_handler() = 0;
        virtual void fd_is_readable(int) = 0;
        virtual void fd_is_writable(int) = 0;
        virtual void read_timeout(int) = 0;
        virtual void write_timeout(int) = 0;
        };
    struct handler_properties
        {
        short        poll_events;
        unsigned int read_timeout;
        unsigned int write_timeout;
        };
    void register_handler(int fd, event_handler& handler,
                          const handler_properties& properties);
    void remove_handler(int fd);
    void schedule();
    };
```

The objective of `scheduler` is to manage a set of file descriptors and corresponding event handlers, and to notify the event handlers every time an event occurs on "their" file descriptor. An "event" is the file descriptor becoming readable or writable as well as a a file descriptor *not* becoming readable or writable within a certain amount of time (timeout).

An "event handler" is anything that satisfies the `scheduler::event_handler` interface. Since all member functions in `event_handler` are pure virtual, the programmer may register a derived version of that class as event handler, thereby providing the appropriate code for the four callback functions.

Registering a file descriptor and its event handler takes place via the `register_handler()` member function, which expects an additional third parameter: the scheduler::handler_properties. That is a structure containing the flags that determine what events the handler is interested in, plus the appropriate timeouts (in seconds) that apply when no event occurred for that amount of time.

Once at least one file descriptor (and its event handler) has been registered, the member function `schedule()` may be called. `schedule()` will use the UNIX function poll(2) to determine the state of the registered file descriptors. When `poll()` returns, `schedule()` will iterate through all registered descriptors and execute the appropriate callbacks depending on whether the descriptor is readable or writable.

Readers who are familiar with the concept of "design patterns" will have recognized this functionality: `scheduler` implements a "reactor" pattern — often referred to as "dispatcher", too.

Since grey is, young friend, all theory, let us take a look at a specific example how `scheduler` might be used in practice. Figure 2 shows a callback that will read from file descriptor 0 and write anything it read to descriptor 1.

Figure 2. An example event handler for the `scheduler` class

```
class my_handler : public scheduler::event_handler
    {
  public:
    my_handler(scheduler& sched) : mysched(sched) { }
    ~my_handler()                                 { }
    scheduler::handler_properties prop;
  private:
    virtual void fd_is_readable(int fd)
        {
        int rc = read(fd, &tmp, sizeof(tmp));
        if (rc == 0)
            mysched.remove_handler(0);
        else if (rc > 0)
            {
```

```
            buffer.append(tmp, rc);
            prop.poll_events = POLLOUT;
            mysched.register_handler(1, *this, prop);
            }
        }
    virtual void fd_is_writable(int fd)
        {
        if (buffer.empty())
            mysched.remove_handler(1);
        int rc = write(fd, buffer.data(), buffer.length());
        if (rc > 0)
            buffer.erase(0, rc);
        }
    char tmp[1024];
    string buffer;
    scheduler& mysched;
    };
```

Not all the code necessary to actually *run* that example is shown here, of course; we cut the program down to the parts essential for understanding the underlying principle. The interested reader will find the complete source code for this example at [Libscheduler], though.

Anyway, assume that the callback has been registered to be called when file descriptor 0 becomes readable. Assume furthermore that we don't care about timeouts. Then `schedule()` will use `poll()` to wait until file descriptor 0 becomes readable and then call the `fd_is_readable()` function of our callback.

The function now reads as many bytes from the descriptor as it can store in its internal buffer. (It won't matter if we don't read all available data in one go; if there is additional data pending on that stream, we'll be called again right away.) The data we read is now appended to a dynamically growing `string` instance, which we use as our output buffer, as will be seen later. Once the data has been appended to that `string`, we register ourself as callback for file descriptor 1, waiting for the "writable" event.

So the `fd_is_readable()` function will go on and on until the data stream associated with file descriptor 0 ends: Then the read(2) call will fail. On that occasion, we simply unregister ourself. Please note that this does not mean the process as a whole is finished now: There might still be some data waiting in the output buffer for descriptor 1 becoming writable!

The `fd_is_writable()` function behaves as follows: It will check whether the output buffer is empty. If it is, it unregisters itself, because obviously there is nothing to be done. If the output buffer is not empty, it will write as much data to file descriptor 1 as it can. Any data that has been written successfully will be removed from the output buffer.

That's it; all the intelligence is in that short class definition shown in Figure 2. The `main()` routine of the program does nothing but instantiate the `scheduler` and `my_handler` class, register the latter as is-readable-callback for file descriptor 0, and call `schedule()` — if you are interested, you can see the complete source code at [Libscheduler].

It is easy to design a state machine that fulfils a certain task like reading, processing, and writing some data — exactly what the majority of FastCGI applications will do —, but using the `scheduler` class, it is even easier to have an arbitrary number of instances of that state machine run simultaneously, serving different channels.

3.2. FastCGI Protocol Driver Library

This link library provides two classes: `FCGIProtocolDriver` and `FCGIRequest`. The protocol driver class is basically an interpreter of the FastCGI protocol, which frees the application developer from worrying about byte counting, endian conversion, and creating or interpreting packet headers. You will usually have one instance of this class per connection to the web server.

The second class, `FCGIRequest`, is spawned by the protocol driver every time a BEGIN_REQUEST packet is received. It contains the decoded information of the FastCGI request, such as a map of the parameters and the contents of the STDIN stream. It also contains a reference to the actual *handler* of the request — the code provided by the application developer.

3.2.1. The `FCGIProtocolDriver` class

The declaration of the public interface of the class is shown in Figure 3.

Figure 3. Declaration of the `FCGIProtocolDriver` class

```
class FCGIProtocolDriver
    {
  public:
    struct OutputCallback
        {
        virtual ~OutputCallback() = 0;
        virtual void operator() (const void*, size_t) = 0;
        };

    FCGIProtocolDriver(OutputCallback& cb);

    void process_input(const void* buf, size_t count);
    FCGIRequest* get_request();
    bool have_active_requests();
    };
```

The purpose of this class is to handle the connection to the web server, to interpret the FastCGI protocol, and to trigger appropriate actions when a new request is received. In order to achieve this objective, the class must read all data sent by the web server and it must be able to send data back to the web server. Despite this, `FCGIProtocolDriver` does not perform any I/O itself! The reason for this is that we wanted the class to be independent of the I/O model chosen by the application developer. The class can be used in combination with the `scheduler` class, it can be used in a multithreaded environment, or it can be used in a multiprocess environment. Obviously, in our opinion, `scheduler` is the way to go because it has the least context-switching overhead and thus best performance, but `FCGIProtocolDriver` does not force that decision upon the developer.

Rather than performing I/O itself, the class expects to have its `process_input` member function called with all data read from the web server. For writing back, it uses the abstract callback functor `OutputCallback`, which has to be provided when `FCGIProtocolDriver` is instantianted. An example architecture for this, based on `scheduler`, will be shown in Section 3.3. Please note that the output callback *must* accept the entire contents of the buffer! If the buffer is too large to be written at once, the callback must buffer the remainder.

Any FastCGI protocol data fed into the `process_input` member function will be interpreted by the class, and whenever a BEGIN_REQUEST packet is received, the class will create a new `FCGIRequest` class instance that is tied to this request. The application developer must call the `get_request` member function regularly in order to obtain that request. More on that topic will follow in the next section.

The only other publically visible member function is `have_active_requests`, which the application developer may call to learn whether there are still active requests associated with this protocol driver — and thus with the connection the protocol driver “listens” to. If this function returns `false`, the program may choose to destroy the protocol driver instance and to close the connection to the web server in order to free up resources.

When a `FCGIProtocolDriver` class instance is destroyed, its destructor will automatically terminate and destroy all `FCGIRequest` instances spawned by this instance as well. The protocol driver will also destroy any `FCGIRequest` instances that throw an exception, assuming that if the exception could have been handled internally, it would have been. The only exception to this rule (no pun intended) is the exception `fcgi_io_callback_error`, which is defined in the `fastcgi.hh` header file: This exception may be thrown by the output callback that has been provided by the application, in order to communicate I/O errors to the application.

`FCGIProtocolDriver` itself may throw the following exceptions:

`unsupported_fcgi_version`

> The current version of libfastcgi supports protocol version 1. Which is the only protocol version defined at this point.

`duplicate_begin_request`

> A BEGIN_REQUEST packet has been received with a duplicate request ID. This is probably a non-recoverable error and the only way to deal with it is to terminate the connection, because it means that the web server’s internals are screwed up.

All exceptions thrown by `FCGIProtocolDriver` are derived from the abstract base class `fcgi_error`, which is in turn derived from `std::runtime_error`.

3.2.2. The `FCGIRequest` class

The declaration of the public interface of the class is shown in Figure 4.

Figure 4. Declaration of the `FCGIRequest` class

```
class FCGIRequest
    {
  public:
    enum role_t { RESPONDER, AUTHORIZER, FILTER };

    const u_int16_t id;
    const role_t role;
    const bool keep_connection;
    bool aborted;
    std::map<std::string,std::string> params;
    std::string stdin_stream, data_stream;
    bool stdin_eof, data_eof;

    enum ostream_type_t { STDOUT, STDERR };
    void write(const std::string& buf, ostream_type_t stream = STDOUT);
    void write(const char* buf, size_t count, ostream_type_t stream = STDOUT);

    enum protocol_status_t
        {
        REQUEST_COMPLETE,
        CANT_MPX_CONN,
        OVERLOADED,
        UNKNOWN_ROLE
        };
    void end_request(u_int32_t appStatus, protocol_status_t protStatus);

    struct handler
        {
        virtual ~handler() { }
        virtual void operator()(FCGIRequest*) = 0;
        };
    handler* handler_cb;
    };
```

One `FCGIRequest` class instance is created per received BEGIN_REQUEST packet, and the data that are received for this request is made available in the class's member variables:

`u_int16_t id`

The request ID assigned to this request.

`enum role_t role`

The role the request is expected to fulfil.

`bool keep_connection`

If this boolean is `true`, the connection is supposed to be kept open even after the request has been dealt with. (It may, of course, be closed nonetheless.) If it is set to `false`, the application *should* close the connection when the request finishes, because the web server will not send a new request on this connection. In fact, some web servers will behave strangely when the connection is not closed in this case. This flag will most likely be interpreted by the I/O framework after it has obtained the request instance by calling `FCGIRequest::get_request`.

`bool aborted`

This boolean will be set to `true` by the protocol driver when an ABORT_REQUEST packet has been received — for example, when the user pressed the "stop" button in his browser. The request handler should always test this flag when it is called and terminate propperly.

`std::map<std::string,std::string> params`

This map will contain the variables defined by the CGI protocol mapped from the variable name to its contents. It may potentially also contain additional variables, depending on the configuration of the web server.

`std::string stdin_stream, data_stream`

These strings will contain the data sent by the web server on the STDIN and DATA stream, respectively. New input will be appended to the end of the string. The application should remove all data it has processed.

`bool stdin_eof, bool data_eof`

These booleans will be set to `true` by the protocol driver when the web server sends the end-of-stream packet for the STDIN and the DATA stream, respectively.

The only member variable left is the *`handler_cb`* pointer, which is initialized to `NULL` by the protocol driver. This field points to a function object that has to be derived from the purely abstract interface `FCGIRequest::handler`, and it may be set by the application when it obtains the request instance from the protocol driver. If this pointer is not `NULL`, the functor it points to will be called every time the `FCGIRequest` changes in any way — for instance, when new stream data have been received, when the request has been aborted, when an end-of-stream packet has been received, etc. . . .

It is important to know that `FCGIRequest`'s destructor will execute `operator delete` on the request handler that is currently registered; thus it is a really good idea not to register callbacks that are constructed on the stack!

The request handler is passed a pointer to "its" request structure as a function parameter, which the handler can use to access the class's contents or to call one of the following member functions:

`write(...)`

This member function will write date to either the STDOUT or the STDERR stream using the output callback of the protocol driver instance that spawned this request.

`end_request(...)`

This member function may be called to terminate the request by the application. If no error occurred and the request is terminated simply because it has been processed completely, application status zero and protocol status `REQUEST_COMPLETE` must be returned. In case of an error, the application and protocol status is implementation defined.

3.3. FastCGI Application Framework

The example program that comes with this paper, and which can be found at [Simons2002], contains the header file `infrastructure.hh`. This file defines a generic framework for FastCGI application developers. It takes care of the low-level socket communication (I/O), error recovery in case of run-time errors, and instantiation and handling of the appropriate `FCGIProtocolDriver` and `FCGIRequest` objects. It provides the following classes:

`template<T> Listener`

A `Listener<T>` object can be constructed with a reference to a `scheduler` object. It will register itself as a `scheduler::event_handler` for the standard input file descriptor and wait for incoming connections on this socket. (This is because Apache's mod_fastcgi starts the configured FastCGI applications with their standard input channel set to a Unix IPC socket. This gives faster data transfers than using a full-blown TCP connection.) Every time a connection comes in, the `Listener<T>` instantiates an object of type `T` and provides a reference to the `scheduler` and the newly accepted socket as parameters to the constructor.

`template<T> ConnectionHandler`

This class template provides all mechanisms required to handle a FastCGI connection to the web server. As a template parameter, it expects the type of a `FCGIRequest::handler` class, which will be instantiated using the default constructor every time a new FastCGI request is received.

A connection handler is instantiated with a reference to a `scheduler` object and a socket descriptor. It will use the scheduler to listen for input on the socket. If the socket becomes readable, the input read is forwarded to the `FCGIProtocolDriver`, which the connection handler contains privately. Then, the connection handler queries the `FCGIProtocolDriver` for potentially newly created `FCGIRequest` instances, and — if one is available — creates an instance of `T` and registers it as the event handler.

Any exception thrown by the protocol driver or by the request handler will lead to the destruction of the connection — and thus the termination of all of its requests. Thus, authors of request handlers are encouraged not to leak any exceptions their code may throw, unless they really don't know any way to handle them. The most sensible thing to do in case of an error is for the request handler to just abort the request by calling the `end_request` method of `FCGIRequest`. This will notify the web server and destroy the request and its request handler, but all other active requests will continue to run.

Figure 5. Declaration of the `Listener<T>` class template

```
template<class T>
class Listener : public scheduler::event_handler
    {
  public:
    Listener(scheduler& sched);
    virtual ~Listener()
    };
```

Figure 6. Declaration of the `ConnectionHandler<T>` class template

```
template<class T>
class ConnectionHandler : public scheduler::event_handler,
                          public FCGIProtocolDriver::OutputCallback
    {
  public:
    ConnectionHandler(scheduler& sched, int sock);
    };
```

3.4. echo.fcgi

This FastCGI application echoes all received information back to the user in an HTML formatted page. This includes:

- The process ID of the application,
- the ID of the FastCGI request,
- the request's PARAM variables, and
- the contents of the DATA stream.

That in itself is arguably not very useful, but hopefully the program is a useful example of how all the components of the FastCGI application framework can be put together.

The echo.fcgi program uses the helper classes described in Section 3.3 to provide its functionality. It consists only of a *very* short `main` function, which constructs the `scheduler` and `Listener` objects, and the class `RequestHandler`, which is derived from `FCGIRequest::handler`.

4. Advantages of the Proposed Architecture

4.1. Application Lifetime Is Detached from Web Pages

One major shortcoming of PHP and alikes is that the developer is forced to think in web pages: The application starts when a web page is requested, and it terminates as soon as the web page has been delivered. But you don't really want to design applications around the user interface; the focus of the design should be on the task the application is supposed to do.

Having a permanently running process makes it easier for the developer to design the application in a more structured way, because he has a clear separation between the program doing the actual work and the mark-up of the results of that work.

But the most important advantage, gained by FastCGI programs outliving the web page, is a different one: You don't lose your context. The application does not need to read its configuration file again and again, it doesn't have to open a connection to the database every time a request comes in, and it certainly doesn't need to pass a myriad of different variables from page to page just because all variables are gone by the time the user requests the next page. One can implement efficient caching, share results between requests, etc.

Numerous advantages result from your application no longer being tied to web pages — most of which will be described in this section. But when reading the following paragraphs, keep in mind that all these optimizations are possible only because FastCGI applications don't lose their context.

4.2. Scalability

FastCGI applications scale easily. One could have several separate machines running the application, using the web server to balance the load between these machines. With the patch available at [Armbruster], the web server will even ensure that subsequent requests from the same user are handed over to the same FastCGI instance every time, which makes life a lot easier if the application needs some user context that would otherwise have to be shared among all running instances.

Alternatively, one might choose not to rely on the web server for the load balancing, but to build a small dispatcher application instead. Such a dispatcher would accept all incoming requests from the web server and forward them to the actual applications according to some clever algorithm. The possibilities are endless. Because the application is inherently capable of communicating over the network, it is easy to distribute the load between multiple servers.

4.3. No Context-Switch Overhead

In Section 2.2 we talked about this point already. "Ordinary" web-based applications, as we know them today, need one process per request. FastCGI applications don't. This allows for more efficient use of the available CPU power and system resources. As a result, FastCGI applications will usually be capable of handling hundreds of simultaneous requests, a number the process-based architecture cannot achieve.

4.4. Caching

Caching is about the most powerful tool to increase the performance of a web-based application. With FastCGI, all incoming requests will go through the same process, so you can cache much more efficiently than you could in CGI or PHP applications. In fact, in those latter architectures, you can hardly cache at all, because the applications terminate after each request. If a request dispatcher as described in Section 4.2 is deployed, it could be used to cache entire requests, too.

There are some high-quality implementations of caching algorithms available on the Internet — C++ programmers might want to look at the “Cache Associative Containers” available at [Cerniglia]. Using this class, adding caching capabilities to your applications is really simple.

What types of information an application should cache depends very much on the application and on the data, so no universally valid recommendation can be given. Here are some ideas, though, on what you might want to look at:

- Build MD5 hashes over all relevant parameters of an incoming request and store the page you produced using that hash as a key. When someone presses the “reload” button, you don’t have to re-create that page. Neither will you have to re-create it when somebody else submits the same request.
- Cache the results of database queries. Talking to your database is about the slowest thing you can do.
- If your application has one of those “the first 50 results on this page, the next 50 results on the next page” things, read all results from the database at once — most databases don’t operate any differently anyway — and precompute *all* pages; then put them into a cache. If the client chooses to display the next page, you won’t have to query the database again.
- If your database is not too big, load the entire database into memory at start-up, and store it in some nifty trees or hash tables, so that you can access it without having to talk to the database.

4.5. C versus PHP/Perl/TCL/. . .

The architecture described in this paper uses the C++ programming language. Of course, it is possible to implement FastCGI applications in any programming language that supports interprocess communication, be it Perl, TCL, or even shell scripts. We nonetheless chose C++, because we feel that “real” programming languages are more appropriate for the implementation of complex applications. Writing web-based applications in C or C++ gives the developer more control over the machine. Especially under the UNIX operating system, numerous mechanisms are available only to software writers who use C.

Another advantage is that writing web-based applications in C or C++ gives you access to billions of lines of code freely available on the Internet, no matter whether it is an arbitrary-precision math library, a library for the creation and manipulition of images, or a set of routines implementing strong cryptography. It’s all there — just use it!

Also, the applications are self-contained; if the latest PHP version has a nasty bug, what do you care? All the bugs in the code are your own bugs — assuming that your operating system has no major shortcomings. (Don’t laugh, there seem to be people on this planet who put up with buggy operating systems!) There are no weird limitations on how much memory your program may use or how long it may run forced upon your process by the run-time system. Heck, there ain’t no such thing as a run-time system!

Last but not least, a good reason for C-language applications is that your code runs faster than any of the interpreted languages.

4.6. Ease of Development

Developing FastCGI applications within the framework described in Section 3 is actually comparatively easy. Once you get used to designing your applications like a state machine, you’ll find that you can program web-based applications almost like an interactive process. In principle, there is no difference: The application receives some input from the user, processes it, writes back some output, and waits for the next event.

Having *a single* process that takes care of all the requests has other unsuspected benefits: record locking and synchronization are easier to implement. One problem every web-based application we know of has is that they can't cope with race conditions: User A requests a page that will enable him to edit the contents of some database fields. Instead of editing the contents and submitting the changes immediately, though, he is interrupted and leaves his terminal with the browser still open. Meanwhile, user B requests the very same page, edits the contents of the fields, and submits his changes. A few minutes later, user A returns to his machine and submits his changes, too, thereby overwriting the changes made by user B.

In FastCGI applications, you can use semaphores, record locks, and other mechanisms to avoid that race condition, because these locks won't be lost when the request is finished — your application doesn't terminate. You can even implement timeouts for locks; something that other web developers can't do trivially.

Finally, writing web-based applications using the FastCGI interface promotes the strict separation of content and layout, because you really don't want to have your HTML pages compiled into the application itself. Instead, build yourself some kind of template mechanism; add magic words to the HTML page that your application will replace with the appropriate data before delivering it.

An even more sophisticated approach would be to have the FastCGI application gather all data required to create the page. Then call PHP and provide a "fake" request, which passes those values to the PHP interpreter via POST. Now all the formatting can be done in PHP and all the processing in the FastCGI application. An ideal combination, because the C programmers don't have to know HTML, JavaScript, etc., and the web designers don't have to know how to program. Most of them don't anyway.

5. Disadvantages of the Proposed Architecture

Nothing is perfect; of course, FastCGI applications do have disadvantages when compared to the traditional architecture:

- Because the application won't terminate after each request, memory leaks are a great threat to the application's stability.
- Since FastCGI applications do usually multiplex numerous requests at the same time, bugs have severe consequences: If a FastCGI application crashes because of a bug, it will take *all* other requests down with it, rather than just the request it was processing when the error occurred.
- The FastCGI interface is way more complex than solutions like PHP. Learning how to write multiplexing FastCGI applications will take some time. We feel that this investment of time pays off rather quickly, but sometimes other people tend to see this differently.

6. Conclusion

The intention of this paper is *not* to denigrate PHP or comparable solutions. We are firm believers in the "right tool for the job" principle, and without a doubt, there are countless jobs for which PHP is just perfect. But PHP was never meant to be used to write complex web-based applications. We think that the FastCGI interface offers some substantial architectural benefits to software engineers, which — combined with the use of a powerful programming language like C — promote a better design and consequently more elegant and better performing web-based applications.

References

[FastCGISpec] *FastCGI Specification (http://www.fastcgi.com/devkit/doc/fcgi-spec.html)*, Mark R. Brown, 1996-04.

[CGISpec] *Common Gateway Interface Specification (http://hoohoo.ncsa.uiuc.edu/cgi/interface.html)*.

[ModFastCGI] *mod_fastcgi 2.2.12 (http://www.fastcgi.com/dist/mod_fastcgi-2.2.12.tar.gz): The FastCGI Interface Module For Apache 1.x*.

[Armbruster] *Armbruster's FastCGI Session Affinity Patch (http://www.tfarmbruster.com/fcgi_sa.htm)*.

[Jitterbug] *Jitterbug (http://jitterbug.samba.org/): A Web-based Bug Tracking System*.

[PHP] *PHP (http://www.php.net/): Hypertext Preprocessor*.

[SourceForge] *SourceForge.Net (http://sourceforge.net/): Breaking Down the Barriers to Open-Source Development*.

[Cerniglia] *The Addition of Cache Associative Containers to STL (http://www.cs.rpi.edu/~harrisod/AP/)*, Stephen Cerniglia, 1996-10-10.

[Libscheduler] *Generic Multiplexing Scheduler Library (http://cryp.to/libscheduler/)*, Peter Simons.

[Libfastcgi] *FastCGI Protocol Driver Library (http://cryp.to/libfastcgi/)*, Peter Simons.

[Simons2002] *FastCGI — The Forgotten Treasure (http://cryp.to/publications/fastcgi/)*.