FastCGI Specification

Mark R. Brown
Open Market, Inc.

Document Version: 1.0
29 April 1996

---

1. Introduction

FastCGI is an open extension to CGI that provides high performance for all Internet applications without the penalties of Web server APIs.

This specification has narrow goal: to specify, from an application perspective, the interface between a FastCGI application and a Web server that supports FastCGI. Many Web server features related to FastCGI, e.g. application management facilities, have nothing to do with the application to Web server interface, and are not described here.

This specification is for Unix (more precisely, for POSIX systems that support Berkeley Sockets). The bulk of the specification is a simple communications protocol that is independent of byte ordering and will extend to other systems.

We'll introduce FastCGI by comparing it with conventional Unix implementations of CGI/1.1. FastCGI is designed to support long-lived application processes, i.e. *application servers*. That's a major difference compared with conventional Unix implementations of CGI/1.1, which construct an application process, use it respond to one request, and have it exit.

The initial state of a FastCGI process is more spartan than the initial state of a CGI/1.1 process, because the FastCGI process doesn't begin life connected to anything. It doesn't have the conventional open files `stdin`, `stdout`, and `stderr`, and it doesn't receive much information through environment variables. The key piece of initial state in a FastCGI process is a listening socket, through which it accepts connections from a Web server.

After a FastCGI process accepts a connection on its listening socket, the process executes a simple protocol to receive and send data. The protocol serves two purposes. First, the protocol multiplexes a single transport connection between several independent FastCGI requests. This supports applications that are able to process concurrent requests using event-driven or multi-threaded programming techniques. Second, within each request the protocol provides several independent data streams in each direction. This way, for instance, both `stdout` and `stderr` data pass over a single transport connection from the application to the Web server, rather than requiring separate pipes as with CGI/1.1.

A FastCGI application plays one of several well-defined *roles*. The most familiar is the *Responder* role, in which the application receives all the information associated with an HTTP request and generates an HTTP response; that's the role CGI/1.1 programs play. A second role is *Authorizer*, in which the application receives all the information associated with an HTTP request and generates an authorized/unauthorized decision. A third role is *Filter*, in which the application receives all the information associated with an

HTTP request, plus an extra stream of data from a file stored on the Web server, and generates a "filtered" version of the data stream as an HTTP response. The framework is extensible so that more FastCGI can be defined later.

In the remainder of this specification the terms "FastCGI application," "application process," or "application server" are abbreviated to "application" whenever that won't cause confusion.

2. Initial Process State

2.1 Argument list

By default the Web server creates an argument list containing a single element, the name of the application, taken to be the last component of the executable's path name. The Web server may provide a way to specify a different application name, or a more elaborate argument list.

Note that the file executed by the Web server might be an interpreter file (a text file that starts with the characters `#!`), in which case the application's argument list is constructed as described in the `execve` manpage.

2.2 File descriptors

The Web server leaves a single file descriptor, `FCGI_LISTENSOCK_FILENO`, open when the application begins execution. This descriptor refers to a listening socket created by the Web server.

`FCGI_LISTENSOCK_FILENO` equals `STDIN_FILENO`. The standard descriptors `STDOUT_FILENO` and `STDERR_FILENO` are closed when the application begins execution. A reliable method for an application to determine whether it was invoked using CGI or FastCGI is to call `getpeername(FCGI_LISTENSOCK_FILENO)`, which returns -1 with `errno` set to `ENOTCONN` for a FastCGI application.

The Web server's choice of reliable transport, Unix stream pipes (`AF_UNIX`) or TCP/IP (`AF_INET`), is implicit in the internal state of the `FCGI_LISTENSOCK_FILENO` socket.

2.3 Environment variables

The Web server may use environment variables to pass parameters to the application. This specification defines one such variable, `FCGI_WEB_SERVER_ADDRS`; we expect more to be defined as the specification evolves. The Web server may provide a way to bind other environment variables, such as the `PATH` variable.

2.4 Other state

The Web server may provide a way to specify other components of an application's initial process state, such as the priority, user ID, group ID, root directory, and working directory of the process.

3. Protocol Basics

3.1 Notation

We use C language notation to define protocol message formats. All structure elements are defined in terms of the `unsigned char` type, and are arranged so that an ISO C compiler lays them out in the obvious manner, with no padding. The first byte defined in the structure is transmitted first, the second byte second, etc.

We use two conventions to abbreviate our definitions.

First, when two adjacent structure components are named identically except for the suffixes "`B1`" and "`B0`," it means that the two components may be viewed as a single number, computed as `B1<<8 + B0`. The name of this single number is the name of the components, minus the suffixes. This convention generalizes in an obvious way to handle numbers represented in more than two bytes.

Second, we extend C `struct`s to allow the form

```
struct {
    unsigned char mumbleLengthB1;
    unsigned char mumbleLengthB0;
    ... /* other stuff */
    unsigned char mumbleData[mumbleLength];
};
```

meaning a structure of varying length, where the length of a component is determined by the values of the indicated earlier component or components.

3.2 Accepting Transport Connections

A FastCGI application calls `accept()` on the socket referred to by file descriptor `FCGI_LISTENSOCK_FILENO` to accept a new transport connection. If the `accept()` succeeds, and the `FCGI_WEB_SERVER_ADDRS` environment variable is bound, the application application immediately performs the following special processing:

- `FCGI_WEB_SERVER_ADDRS`: The value is a list of valid IP addresses for the Web server.

  If `FCGI_WEB_SERVER_ADDRS` was bound, the application checks the peer IP address of the new connection for membership in the list. If the check fails (including the possibility that the connection didn't use TCP/IP transport), the application responds by closing the connection.

`FCGI_WEB_SERVER_ADDRS` is expressed as a comma-separated list of IP addresses. Each IP address is written as four decimal numbers in the range [0..255] separated by decimal points. So one legal binding for this variable is `FCGI_WEB_SERVER_ADDRS=199.170.183.28,199.170.183.71`.

An application may accept several concurrent transport connections, but it need not do so.

3.3 Records

Applications execute requests from a Web server using a simple protocol. Details of the protocol depend upon the application's role, but roughly speaking the Web server first sends parameters and other data to the application, then the application sends result data to the Web server, and finally the application sends the Web server an indication that the request is complete.

All data that flows over the transport connection is carried in *FastCGI records*. FastCGI records accomplish two things. First, records multiplex the transport connection between several independent FastCGI requests. This multiplexing supports applications that are able to process concurrent requests using event-driven or multi-threaded programming techniques. Second, records provide several independent data streams in each direction within a single request. This way, for instance, both `stdout` and `stderr` data can pass over a single transport connection from the application to the Web server, rather than requiring separate connections.

```
typedef struct {
    unsigned char version;
    unsigned char type;
    unsigned char requestIdB1;
    unsigned char requestIdB0;
    unsigned char contentLengthB1;
    unsigned char contentLengthB0;
    unsigned char paddingLength;
    unsigned char reserved;
    unsigned char contentData[contentLength];
    unsigned char paddingData[paddingLength];
} FCGI_Record;
```

A FastCGI record consists of a fixed-length prefix followed by a variable number of content and padding bytes. A record contains seven components:

- `version`: Identifies the FastCGI protocol version. This specification documents `FCGI_VERSION_1`.
- `type`: Identifies the FastCGI record type, i.e. the general function that the record performs. Specific record types and their functions are detailed in later sections.
- `requestId`: Identifies the *FastCGI request* to which the record belongs.

- `contentLength`: The number of bytes in the `contentData` component of the record.
- `paddingLength`: The number of bytes in the `paddingData` component of the record.
- `contentData`: Between 0 and 65535 bytes of data, interpreted according to the record type.
- `paddingData`: Between 0 and 255 bytes of data, which are ignored.

We use a relaxed C `struct` initializer syntax to specify constant FastCGI records. We omit the `version` component, ignore padding, and treat `requestId` as a number. Thus `{FCGI_END_REQUEST, 1, {FCGI_REQUEST_COMPLETE,0}}` is a record with `type == FCGI_END_REQUEST`, `requestId == 1`, and `contentData == {FCGI_REQUEST_COMPLETE,0}`.

Padding

The protocol allows senders to pad the records they send, and requires receivers to interpret the `paddingLength` and skip the `paddingData`. Padding allows senders to keep data aligned for more efficient processing. Experience with the X window system protocols shows the performance benefit of such alignment.

We recommend that records be placed on boundaries that are multiples of eight bytes. The fixed-length portion of a `FCGI_Record` is eight bytes.

Managing Request IDs

The Web server re-uses FastCGI request IDs; the application keeps track of the current state of each request ID on a given transport connection. A request ID `R` becomes active when the application receives a record `{FCGI_BEGIN_REQUEST, R, ...}` and becomes inactive when the application sends a record `{FCGI_END_REQUEST, R, ...}` to the Web server.

While a request ID `R` is inactive, the application ignores records with `requestId == R`, except for `FCGI_BEGIN_REQUEST` records as just described.

The Web server attempts to keep FastCGI request IDs small. That way the application can keep track of request ID states using a short array rather than a long array or a hash table. An application also has the option of accepting only one request at a time. In this case the application simply checks incoming `requestId` values against the current request ID.

Types of Record Types

There are two useful ways of classifying FastCGI record types.

The first distinction is between *management* records and *application* records. A management record contains information that is not specific to any Web server request, such as information about the protocol capabilities of the application. An application record contains information about a particular request, identified by the `requestId` component.

Management records have a `requestId` value of zero, also called the *null request ID*. Application records have a nonzero `requestId`.

The second distinction is between *discrete* and *stream* records. A discrete record contains a meaningful unit of data all by itself. A stream record is part of a *stream*, i.e. a series of zero or more non-empty records (`length != 0`) of the stream type, followed by an empty record (`length == 0`) of the stream type. The `contentData` components of a stream's records, when concatenated, form a byte sequence; this byte sequence is the value of the stream. Therefore the value of a stream is independent of how many records it contains or how its bytes are divided among the non-empty records.

These two classifications are independent. Among the record types defined in this version of the FastCGI protocol, all management record types are also discrete record types, and nearly all application record types are stream record types. But three application record types are discrete, and nothing prevents defining a management record type that's a stream in some later version of the protocol.

3.4 Name-Value Pairs

In many of their roles, FastCGI applications need to read and write varying numbers of variable-length values. So it is useful to adopt a standard format for encoding a name-value pair.

FastCGI transmits a name-value pair as the length of the name, followed by the length of the value, followed by the name, followed by the value. Lengths of 127 bytes and less can be encoded in one byte, while longer lengths are always encoded in four bytes:

```
typedef struct {
    unsigned char nameLengthB0;  /* nameLengthB0  >> 7 == 0 */
    unsigned char valueLengthB0; /* valueLengthB0 >> 7 == 0 */
    unsigned char nameData[nameLength];
    unsigned char valueData[valueLength];
} FCGI_NameValuePair11;

typedef struct {
    unsigned char nameLengthB0;  /* nameLengthB0  >> 7 == 0 */
    unsigned char valueLengthB3; /* valueLengthB3 >> 7 == 1 */
    unsigned char valueLengthB2;
    unsigned char valueLengthB1;
    unsigned char valueLengthB0;
    unsigned char nameData[nameLength];
    unsigned char valueData[valueLength
```

```
            ((B3 & 0x7f) << 24) + (B2 << 16) + (B1 << 8) + B0];
} FCGI_NameValuePair14;

typedef struct {
    unsigned char nameLengthB3;  /* nameLengthB3  >> 7 == 1 */
    unsigned char nameLengthB2;
    unsigned char nameLengthB1;
    unsigned char nameLengthB0;
    unsigned char valueLengthB0; /* valueLengthB0 >> 7 == 0 */
    unsigned char nameData[nameLength
            ((B3 & 0x7f) << 24) + (B2 << 16) + (B1 << 8) + B0];
    unsigned char valueData[valueLength];
} FCGI_NameValuePair41;

typedef struct {
    unsigned char nameLengthB3;  /* nameLengthB3  >> 7 == 1 */
    unsigned char nameLengthB2;
    unsigned char nameLengthB1;
    unsigned char nameLengthB0;
    unsigned char valueLengthB3; /* valueLengthB3 >> 7 == 1 */
    unsigned char valueLengthB2;
    unsigned char valueLengthB1;
    unsigned char valueLengthB0;
    unsigned char nameData[nameLength
            ((B3 & 0x7f) << 24) + (B2 << 16) + (B1 << 8) + B0];
    unsigned char valueData[valueLength
            ((B3 & 0x7f) << 24) + (B2 << 16) + (B1 << 8) + B0];
} FCGI_NameValuePair44;
```

The high-order bit of the first byte of a length indicates the length's encoding. A high-order zero implies a one-byte encoding, a one a four-byte encoding.

This name-value pair format allows the sender to transmit binary values without additional encoding, and enables the receiver to allocate the correct amount of storage immediately even for large values.

3.5 Closing Transport Connections

The Web server controls the lifetime of transport connections. The Web server can close a connection when no requests are active. Or the Web server can delegate close authority to the application (see `FCGI_BEGIN_REQUEST`). In this case the application closes the connection at the end of a specified request.

This flexibility accommodates a variety of application styles. Simple applications will process one request at a time and accept a new transport connection for each request. More complex applications will process concurrent requests, over one or multiple transport connections, and will keep transport connections open for long periods of time.

A simple application gets a significant performance boost by closing the transport connection when it has finished writing its response. The Web server needs to control the connection lifetime for long-lived connections.

When an application closes a connection or finds that a connection has

closed, the application initiates a new connection.

4. Management Record Types

4.1 FCGI_GET_VALUES, FCGI_GET_VALUES_RESULT

The Web server can query specific variables within the application. The server will typically perform a query on application startup in order to to automate certain aspects of system configuration.

The application receives a query as a record `{FCGI_GET_VALUES, 0, ...}`. The `contentData` portion of a `FCGI_GET_VALUES` record contains a sequence of name-value pairs with empty values.

The application responds by sending a record `{FCGI_GET_VALUES_RESULT, 0, ...}` with the values supplied. If the application doesn't understand a variable name that was included in the query, it omits that name from the response.

`FCGI_GET_VALUES` is designed to allow an open-ended set of variables. The initial set provides information to help the server perform application and connection management:

- `FCGI_MAX_CONNS`: The maximum number of concurrent transport connections this application will accept, e.g. `"1"` or `"10"`.
- `FCGI_MAX_REQS`: The maximum number of concurrent requests this application will accept, e.g. `"1"` or `"50"`.
- `FCGI_MPXS_CONNS`: `"0"` if this application does not multiplex connections (i.e. handle concurrent requests over each connection), `"1"` otherwise.

An application may receive a `FCGI_GET_VALUES` record at any time. The application's response should not involve the application proper but only the FastCGI library.

4.2 FCGI_UNKNOWN_TYPE

The set of management record types is likely to grow in future versions of this protocol. To provide for this evolution, the protocol includes the `FCGI_UNKNOWN_TYPE` management record. When an application receives a management record whose type `T` it does not understand, the application responds with `{FCGI_UNKNOWN_TYPE, 0, {T}}`.

The `contentData` component of a `FCGI_UNKNOWN_TYPE` record has the form:

```
typedef struct {
    unsigned char type;
    unsigned char reserved[7];
} FCGI_UnknownTypeBody;
```

The `type` component is the type of the unrecognized management record.

5. Application Record Types

5.1 `FCGI_BEGIN_REQUEST`

The Web server sends a `FCGI_BEGIN_REQUEST` record to start a request.

The `contentData` component of a `FCGI_BEGIN_REQUEST` record has the form:

```
typedef struct {
    unsigned char roleB1;
    unsigned char roleB0;
    unsigned char flags;
    unsigned char reserved[5];
} FCGI_BeginRequestBody;
```

The `role` component sets the role the Web server expects the application to play. The currently-defined roles are:

- `FCGI_RESPONDER`
- `FCGI_AUTHORIZER`
- `FCGI_FILTER`

Roles are described in more detail in Section 6 below.

The `flags` component contains a bit that controls connection shutdown:

- `flags & FCGI_KEEP_CONN`: If zero, the application closes the connection after responding to this request. If not zero, the application does not close the connection after responding to this request; the Web server retains responsibility for the connection.

5.2 Name-Value Pair Stream: `FCGI_PARAMS`

`FCGI_PARAMS` is a stream record type used in sending name-value pairs from the Web server to the application. The name-value pairs are sent down the stream one after the other, in no specified order.

5.3 Byte Streams: `FCGI_STDIN`, `FCGI_DATA`, `FCGI_STDOUT`, `FCGI_STDERR`

`FCGI_STDIN` is a stream record type used in sending arbitrary data from the Web server to the application. `FCGI_DATA` is a second stream record type used to send additional data to the application.

`FCGI_STDOUT` and `FCGI_STDERR` are stream record types for sending arbitrary data and error data respectively from the application to the Web server.

5.4 `FCGI_ABORT_REQUEST`

The Web server sends a `FCGI_ABORT_REQUEST` record to abort a request. After receiving `{FCGI_ABORT_REQUEST, R}`, the application responds as soon as possible with `{FCGI_END_REQUEST, R, {FCGI_REQUEST_COMPLETE, appStatus}}`. This is truly a response from the application, not a low-level acknowledgement from the FastCGI library.

A Web server aborts a FastCGI request when an HTTP client closes its transport connection while the FastCGI request is running on behalf of that client. The situation may seem unlikely; most FastCGI requests will have short response times, with the Web server providing output buffering if the client is slow. But the FastCGI application may be delayed communicating with another system, or performing a server push.

When a Web server is not multiplexing requests over a transport connection, the Web server can abort a request by closing the request's transport connection. But with multiplexed requests, closing the transport connection has the unfortunate effect of aborting *all* the requests on the connection.

5.5 `FCGI_END_REQUEST`

The application sends a `FCGI_END_REQUEST` record to terminate a request, either because the application has processed the request or because the application has rejected the request.

The `contentData` component of a `FCGI_END_REQUEST` record has the form:

```
typedef struct {
    unsigned char appStatusB3;
    unsigned char appStatusB2;
    unsigned char appStatusB1;
    unsigned char appStatusB0;
    unsigned char protocolStatus;
    unsigned char reserved[3];
} FCGI_EndRequestBody;
```

The `appStatus` component is an application-level status code. Each role documents its usage of `appStatus`.

The `protocolStatus` component is a protocol-level status code; the possible `protocolStatus` values are:

- `FCGI_REQUEST_COMPLETE`: normal end of request.
- `FCGI_CANT_MPX_CONN`: rejecting a new request. This happens when a Web server sends concurrent requests over one connection to an application that is designed to process one request at a time per connection.
- `FCGI_OVERLOADED`: rejecting a new request. This happens when the application runs out of some resource, e.g. database connections.
- `FCGI_UNKNOWN_ROLE`: rejecting a new request. This happens when the Web

server has specified a role that is unknown to the application.

6. Roles

6.1 Role Protocols

Role protocols only include records with application record types. They transfer essentially all data using streams.

To make the protocols reliable and to simplify application programming, role protocols are designed to use *nearly sequential marshalling*. In a protocol with strictly sequential marshalling, the application receives its first input, then its second, etc. until it has received them all. Similarly, the application sends its first output, then its second, etc. until it has sent them all. Inputs are not interleaved with each other, and outputs are not interleaved with each other.

The sequential marshalling rule is too restrictive for some FastCGI roles, because CGI programs can write to both `stdout` and `stderr` without timing restrictions. So role protocols that use both `FCGI_STDOUT` and `FCGI_STDERR` allow these two streams to be interleaved.

All role protocols use the `FCGI_STDERR` stream just the way `stderr` is used in conventional applications programming: to report application-level errors in an intelligible way. Use of the `FCGI_STDERR` stream is always optional. If an application has no errors to report, it sends either no `FCGI_STDERR` records or one zero-length `FCGI_STDERR` record.

When a role protocol calls for transmitting a stream other than `FCGI_STDERR`, at least one record of the stream type is always transmitted, even if the stream is empty.

Again in the interests of reliable protocols and simplified application programming, role protocols are designed to be *nearly request-response*. In a truly request-response protocol, the application receives all of its input records before sending its first output record. Request-response protocols don't allow pipelining.

The request-response rule is too restrictive for some FastCGI roles; after all, CGI programs aren't restricted to read all of `stdin` before starting to write `stdout`. So some role protocols allow that specific possibility. First the application receives all of its inputs except for a final stream input. As the application begins to receive the final stream input, it can begin writing its output.

When a role protocol uses `FCGI_PARAMS` to transmit textual values, such as the values that CGI programs obtain from environment variables, the length of the value does not include the terminating null byte, and the value itself does

not include a null byte. An application that needs to provide `environ(7)` format name-value pairs must insert an equal sign between the name and value and append a null byte after the value.

Role protocols do not support the non-parsed header feature of CGI. FastCGI applications set response status using the `Status` and `Location` CGI headers.

6.2 Responder

A Responder FastCGI application has the same purpose as a CGI/1.1 program: It receives all the information associated with an HTTP request and generates an HTTP response.

It suffices to explain how each element of CGI/1.1 is emulated by a Responder:

- The Responder application receives CGI/1.1 environment variables from the Web server over `FCGI_PARAMS`.
- Next the Responder application receives CGI/1.1 `stdin` data from the Web server over `FCGI_STDIN`. The application receives at most `CONTENT_LENGTH` bytes from this stream before receiving the end-of-stream indication. (The application receives less than `CONTENT_LENGTH` bytes only if the HTTP client fails to provide them, e.g. because the client crashed.)
- The Responder application sends CGI/1.1 `stdout` data to the Web server over `FCGI_STDOUT`, and CGI/1.1 `stderr` data over `FCGI_STDERR`. The application sends these concurrently, not one after the other. The application must wait to finish reading `FCGI_PARAMS` before it begins writing `FCGI_STDOUT` and `FCGI_STDERR`, but it needn't finish reading from `FCGI_STDIN` before it begins writing these two streams.
- After sending all its `stdout` and `stderr` data, the Responder application sends a `FCGI_END_REQUEST` record. The application sets the `protocolStatus` component to `FCGI_REQUEST_COMPLETE` and the `appStatus` component to the status code that the CGI program would have returned via the `exit` system call.

A Responder performing an update, e.g. implementing a `POST` method, should compare the number of bytes received on `FCGI_STDIN` with `CONTENT_LENGTH` and abort the update if the two numbers are not equal.

6.3 Authorizer

An Authorizer FastCGI application receives all the information associated with an HTTP request and generates an authorized/unauthorized decision. In case of an authorized decision the Authorizer can also associate name-value pairs with the HTTP request; when giving an unauthorized decision the

Authorizer sends a complete response to the HTTP client.

Since CGI/1.1 defines a perfectly good way to represent the information associated with an HTTP request, Authorizers use the same representation:

- The Authorizer application receives HTTP request information from the Web server on the `FCGI_PARAMS` stream, in the same format as a Responder. The Web server does not send `CONTENT_LENGTH`, `PATH_INFO`, `PATH_TRANSLATED`, and `SCRIPT_NAME` headers.
- The Authorizer application sends `stdout` and `stderr` data in the same manner as a Responder. The CGI/1.1 response status specifies the disposition of the request. If the application sends status 200 (OK), the Web server allows access. Depending upon its configuration the Web server may proceed with other access checks, including requests to other Authorizers.

  An Authorizer application's 200 response may include headers whose names are prefixed with `Variable-`. These headers communicate name-value pairs from the application to the Web server. For instance, the response header

  ```
  Variable-AUTH_METHOD: database lookup
  ```

  transmits the value `"database lookup"` with name `AUTH-METHOD`. The server associates such name-value pairs with the HTTP request and includes them in subsequent CGI or FastCGI requests performed in processing the HTTP request. When the application gives a 200 response, the server ignores response headers whose names aren't prefixed with `Variable-` prefix, and ignores any response content.

  For Authorizer response status values other than "200" (OK), the Web server denies access and sends the response status, headers, and content back to the HTTP client.

6.4 Filter

A Filter FastCGI application receives all the information associated with an HTTP request, plus an extra stream of data from a file stored on the Web server, and generates a "filtered" version of the data stream as an HTTP response.

A Filter is similar in functionality to a Responder that takes a data file as a parameter. The difference is that with a Filter, both the data file and the Filter itself can be access controlled using the Web server's access control mechanisms, while a Responder that takes the name of a data file as a parameter must perform its own access control checks on the data file.

The steps taken by a Filter are similar to those of a Responder. The server

presents the Filter with environment variables first, then standard input (normally form `POST` data), finally the data file input:

- Like a Responder, the Filter application receives name-value pairs from the Web server over `FCGI_PARAMS`. Filter applications receive two Filter-specific variables: `FCGI_DATA_LAST_MOD` and `FCGI_DATA_LENGTH`.
- Next the Filter application receives CGI/1.1 `stdin` data from the Web server over `FCGI_STDIN`. The application receives at most `CONTENT_LENGTH` bytes from this stream before receiving the end-of-stream indication. (The application receives less than `CONTENT_LENGTH` bytes only if the HTTP client fails to provide them, e.g. because the client crashed.)
- Next the Filter application receives the file data from the Web server over `FCGI_DATA`. This file's last modification time (expressed as an integer number of seconds since the epoch January 1, 1970 UTC) is `FCGI_DATA_LAST_MOD`; the application may consult this variable and respond from a cache without reading the file data. The application reads at most `FCGI_DATA_LENGTH` bytes from this stream before receiving the end-of-stream indication.
- The Filter application sends CGI/1.1 `stdout` data to the Web server over `FCGI_STDOUT`, and CGI/1.1 `stderr` data over `FCGI_STDERR`. The application sends these concurrently, not one after the other. The application must wait to finish reading `FCGI_STDIN` before it begins writing `FCGI_STDOUT` and `FCGI_STDERR`, but it needn't finish reading from `FCGI_DATA` before it begins writing these two streams.
- After sending all its `stdout` and `stderr` data, the application sends a `FCGI_END_REQUEST` record. The application sets the `protocolStatus` component to `FCGI_REQUEST_COMPLETE` and the `appStatus` component to the status code that a similar CGI program would have returned via the `exit` system call.

A Filter should compare the number of bytes received on `FCGI_STDIN` with `CONTENT_LENGTH` and on `FCGI_DATA` with `FCGI_DATA_LENGTH`. If the numbers don't match and the Filter is a query, the Filter response should provide an indication that data is missing. If the numbers don't match and the Filter is an update, the Filter should abort the update.

7. Errors

A FastCGI application exits with zero status to indicate that it terminated on purpose, e.g. in order to perform a crude form of garbage collection. A FastCGI application that exits with nonzero status is assumed to have crashed. How a Web server or other application manager responds to applications that exit with zero or nonzero status is outside the scope of this specification.

A Web server can request that a FastCGI application exit by sending it `SIGTERM`. If the application ignores `SIGTERM` the Web server can resort to `SIGKILL`.

FastCGI applications report application-level errors with the `FCGI_STDERR` stream and the `appStatus` component of the `FCGI_END_REQUEST` record. In many cases an error will be reported directly to the user via the `FCGI_STDOUT` stream.

On Unix, applications report lower-level errors, including FastCGI protocol errors and syntax errors in FastCGI environment variables, to `syslog`. Depending upon the severity of the error, the application may either continue or exit with nonzero status.

8. Types and Constants

```
/*
 * Listening socket file number
 */
#define FCGI_LISTENSOCK_FILENO 0

typedef struct {
    unsigned char version;
    unsigned char type;
    unsigned char requestIdB1;
    unsigned char requestIdB0;
    unsigned char contentLengthB1;
    unsigned char contentLengthB0;
    unsigned char paddingLength;
    unsigned char reserved;
} FCGI_Header;

/*
 * Number of bytes in a FCGI_Header.  Future versions of the protocol
 * will not reduce this number.
 */
#define FCGI_HEADER_LEN  8

/*
 * Value for version component of FCGI_Header
 */
#define FCGI_VERSION_1           1

/*
 * Values for type component of FCGI_Header
 */
#define FCGI_BEGIN_REQUEST       1
#define FCGI_ABORT_REQUEST       2
#define FCGI_END_REQUEST         3
#define FCGI_PARAMS              4
#define FCGI_STDIN               5
#define FCGI_STDOUT              6
#define FCGI_STDERR              7
#define FCGI_DATA                8
#define FCGI_GET_VALUES          9
#define FCGI_GET_VALUES_RESULT  10
#define FCGI_UNKNOWN_TYPE       11
#define FCGI_MAXTYPE (FCGI_UNKNOWN_TYPE)
```

```
/*
 * Value for requestId component of FCGI_Header
 */
#define FCGI_NULL_REQUEST_ID     0

typedef struct {
    unsigned char roleB1;
    unsigned char roleB0;
    unsigned char flags;
    unsigned char reserved[5];
} FCGI_BeginRequestBody;

typedef struct {
    FCGI_Header header;
    FCGI_BeginRequestBody body;
} FCGI_BeginRequestRecord;

/*
 * Mask for flags component of FCGI_BeginRequestBody
 */
#define FCGI_KEEP_CONN  1

/*
 * Values for role component of FCGI_BeginRequestBody
 */
#define FCGI_RESPONDER  1
#define FCGI_AUTHORIZER 2
#define FCGI_FILTER     3

typedef struct {
    unsigned char appStatusB3;
    unsigned char appStatusB2;
    unsigned char appStatusB1;
    unsigned char appStatusB0;
    unsigned char protocolStatus;
    unsigned char reserved[3];
} FCGI_EndRequestBody;

typedef struct {
    FCGI_Header header;
    FCGI_EndRequestBody body;
} FCGI_EndRequestRecord;

/*
 * Values for protocolStatus component of FCGI_EndRequestBody
 */
#define FCGI_REQUEST_COMPLETE 0
#define FCGI_CANT_MPX_CONN    1
#define FCGI_OVERLOADED       2
#define FCGI_UNKNOWN_ROLE     3

/*
 * Variable names for FCGI_GET_VALUES / FCGI_GET_VALUES_RESULT records
 */
#define FCGI_MAX_CONNS  "FCGI_MAX_CONNS"
#define FCGI_MAX_REQS   "FCGI_MAX_REQS"
#define FCGI_MPXS_CONNS "FCGI_MPXS_CONNS"
```

```
typedef struct {
    unsigned char type;
    unsigned char reserved[7];
} FCGI_UnknownTypeBody;

typedef struct {
    FCGI_Header header;
    FCGI_UnknownTypeBody body;
} FCGI_UnknownTypeRecord;
```

9. References

National Center for Supercomputer Applications, The Common Gateway Interface, version CGI/1.1.

D.R.T. Robinson, The WWW Common Gateway Interface Version 1.1, Internet-Draft, 15 February 1996.

A. Table: Properties of the record types

The following chart lists all of the record types and indicates these properties of each:

- `WS->App`: records of this type can only be sent by the Web server to the application. Records of other types can only be sent by the application to the Web server.
- `management`: records of this type contain information that is not specific to a Web server request, and use the null request ID. Records of other types contain request-specific information, and cannot use the null request ID.
- `stream`: records of this type form a stream, terminated by a record with empty `contentData`. Records of other types are discrete; each carries a meaningful unit of data.

	WS->App	management	stream
FCGI_GET_VALUES	x	x	
FCGI_GET_VALUES_RESULT		x	
FCGI_UNKNOWN_TYPE		x	
FCGI_BEGIN_REQUEST	x		
FCGI_ABORT_REQUEST	x		
FCGI_END_REQUEST			
FCGI_PARAMS	x		x
FCGI_STDIN	x		x
FCGI_DATA	x		x
FCGI_STDOUT			x
FCGI_STDERR			x

B. Typical Protocol Message Flow

Additional notational conventions for the examples:

- The `contentData` of stream records (`FCGI_PARAMS`, `FCGI_STDIN`, `FCGI_STDOUT`, and `FCGI_STDERR`) is represented as a character string. A string ending in `" ... "` is too long to display, so only a prefix is shown.
- Messages sent to the Web server are indented with respect to messages received from the Web server.
- Messages are shown in the time sequence experienced by the application.

1. A simple request with no data on `stdin`, and a successful response:

```
{FCGI_BEGIN_REQUEST,   1, {FCGI_RESPONDER, 0}}
{FCGI_PARAMS,          1, "\013\002SERVER_PORT80\013\016SERVER_ADDR199.170.183.42 ... "}
{FCGI_PARAMS,          1, ""}
{FCGI_STDIN,           1, ""}

    {FCGI_STDOUT,      1, "Content-type: text/html\r\n\r\n<html>\n<head> ... "}
    {FCGI_STDOUT,      1, ""}
    {FCGI_END_REQUEST, 1, {0, FCGI_REQUEST_COMPLETE}}
```

2. Similar to example 1, but this time with data on `stdin`. The Web server chooses to send the parameters using more `FCGI_PARAMS` records than before:

```
{FCGI_BEGIN_REQUEST,   1, {FCGI_RESPONDER, 0}}
{FCGI_PARAMS,          1, "\013\002SERVER_PORT80\013\016SER"}
{FCGI_PARAMS,          1, "VER_ADDR199.170.183.42 ... "}
{FCGI_PARAMS,          1, ""}
{FCGI_STDIN,           1, "quantity=100&item=3047936"}
{FCGI_STDIN,           1, ""}

    {FCGI_STDOUT,      1, "Content-type: text/html\r\n\r\n<html>\n<head> ... "}
    {FCGI_STDOUT,      1, ""}
    {FCGI_END_REQUEST, 1, {0, FCGI_REQUEST_COMPLETE}}
```

3. Similar to example 1, but this time the application detects an error. The application logs a message to `stderr`, returns a page to the client, and returns non-zero exit status to the Web server. The application chooses to send the page using more `FCGI_STDOUT` records:

```
{FCGI_BEGIN_REQUEST,   1, {FCGI_RESPONDER, 0}}
{FCGI_PARAMS,          1, "\013\002SERVER_PORT80\013\016SERVER_ADDR199.170.183.42 ... "}
{FCGI_PARAMS,          1, ""}
{FCGI_STDIN,           1, ""}

    {FCGI_STDOUT,      1, "Content-type: text/html\r\n\r\n<ht"}
    {FCGI_STDERR,      1, "config error: missing SI_UID\n"}
    {FCGI_STDOUT,      1, "ml>\n<head> ... "}
    {FCGI_STDOUT,      1, ""}
    {FCGI_STDERR,      1, ""}
    {FCGI_END_REQUEST, 1, {938, FCGI_REQUEST_COMPLETE}}
```

4. Two instances of example 1, multiplexed onto a single connection. The first

request is more difficult than the second, so the application finishes the requests out of order:

```
{FCGI_BEGIN_REQUEST,   1, {FCGI_RESPONDER, FCGI_KEEP_CONN}}
{FCGI_PARAMS,          1, "\013\002SERVER_PORT80\013\016SERVER_ADDR199.170.183.42 ... "}
{FCGI_PARAMS,          1, ""}
{FCGI_BEGIN_REQUEST,   2, {FCGI_RESPONDER, FCGI_KEEP_CONN}}
{FCGI_PARAMS,          2, "\013\002SERVER_PORT80\013\016SERVER_ADDR199.170.183.42 ... "}
{FCGI_STDIN,           1, ""}

    {FCGI_STDOUT,      1, "Content-type: text/html\r\n\r\n"}

{FCGI_PARAMS,          2, ""}
{FCGI_STDIN,           2, ""}

    {FCGI_STDOUT,      2, "Content-type: text/html\r\n\r\n<html>\n<head> ... "}
    {FCGI_STDOUT,      2, ""}
    {FCGI_END_REQUEST, 2, {0, FCGI_REQUEST_COMPLETE}}
    {FCGI_STDOUT,      1, "<html>\n<head> ... "}
    {FCGI_STDOUT,      1, ""}
    {FCGI_END_REQUEST, 1, {0, FCGI_REQUEST_COMPLETE}}
```
