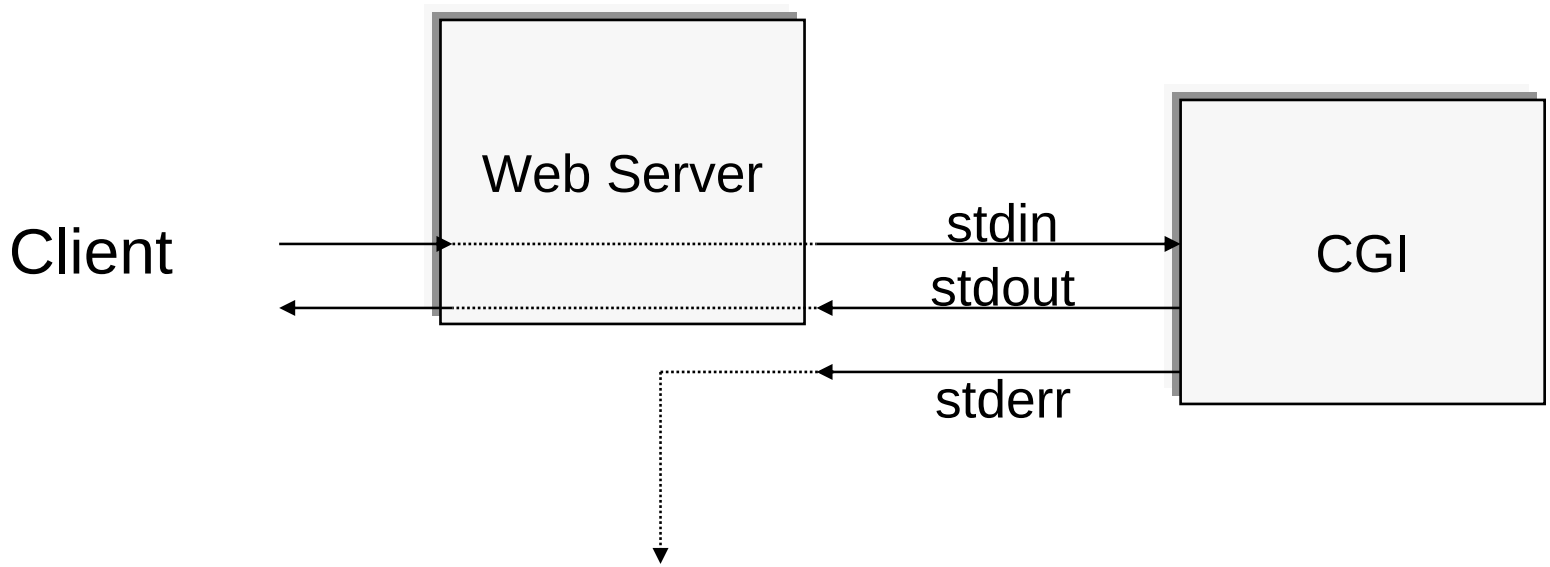
FastCGI

- References:
  - White paper
    - http://www.fastcgi.com/devkit/doc/fastcgi-whitepaper/fastcgi.htm
  - FastCGI Specification
    - http://www.fastcgi.com/devkit/doc/fcgi-spec.html
  - The FastCGI Home Page
    - http://www.fastcgi.com

Web Server - CGI Relationship



Example of CGI

```
main() {
  printf("Content-type: text/html\n\n");
  printf("<html>");
  printf("<body>");
  printf("<h2>Server %s</h2>", getenv(“SERVER_HOSTNAME”));
  printf("</body>");
  printf("</html>");
}
```

Benefits of CGI

- Simplicity.
  - It is easy to understand.
- Language independence.
  - CGI applications can be written in nearly any language.
- Process isolation.
  - Since applications run in separate processes, buggy applications cannot crash the Web server or access the server's private internal state.
- Open standard.
  - Some form of CGI has been implemented on every Web server.
- Architecture independence.
  - CGI is not tied to any particular server architecture (single threaded, multi-threaded, etc.).

Drawbacks of CGI

- Inefficient.
  - Need fork(), exec(), and free the resources.
- Not persistent.
  - Only support a simple “responder”.

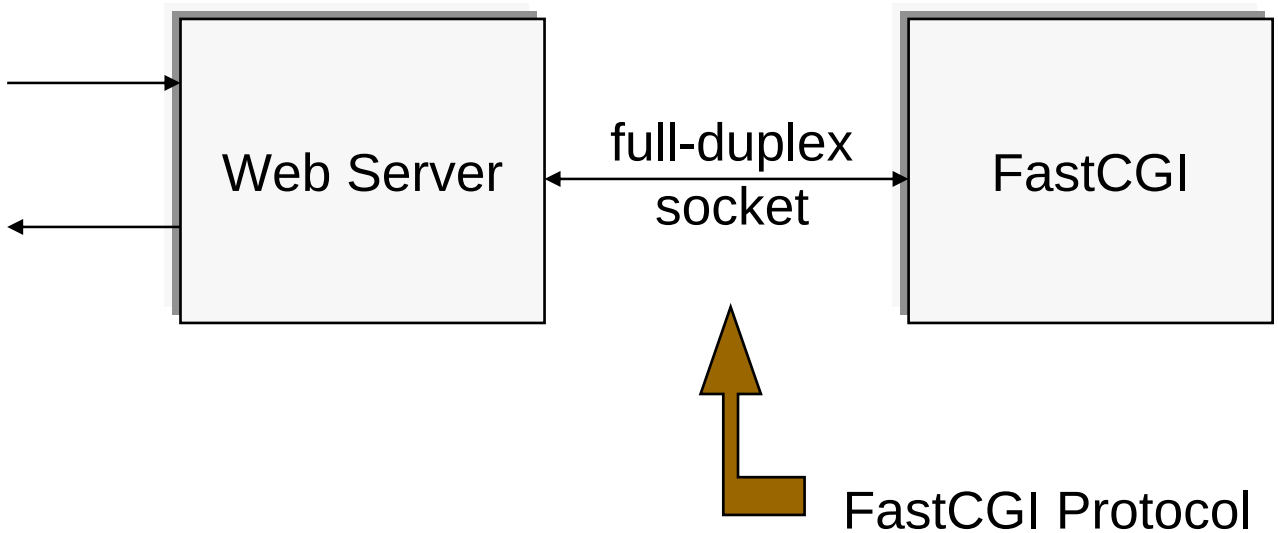
Server APIs

- To improve performance, many vendors develop APIs for their servers.
- Examples
  - NSAPI (by Netscape)
  - ISAPI (by MS Windows)
- Problems:
  - Complexity.
    - Vendor APIs introduce a steep learning curve.
  - Language dependence.
  - No process isolation.
    - Since the applications run in the server's address space, buggy applications can corrupt the core server (or each other).
  - Proprietary.
    - Coding your application to a particular API locks you into a particular vendor's server.
  - Tie-in to server architecture.
    - API applications have to share the same architecture as the server: If the Web server is multi-threaded, the application has to be thread-safe.

FastCGI

- A language and server independent, scalable, open extension to CGI that provides high performance and persistence
- A protocol for data interchange between a web server and a FastCGI application
- The set of libraries that implement the protocol

Web Server - FastCGI Relationship



Differences Between CGI and FastCGI

- FastCGI processes are persistent:
  - After finishing a request, they wait for a new request instead of exiting.
- Instead of using operating system environment variables and pipes, the FastCGI protocol multiplexes the environment information, standard input, output and error over a single full-duplex connection.
  - This allows FastCGI programs to run on remote machines, using TCP connections between the Web server and the FastCGI application.

Advantages of FastCGI

- Performance.
- Simplicity, with easy migration from CGI.
- Language independence.
- Process isolation.
- Non-proprietary.
- Architecture independence.
- Support for distributed computing.

Procedure of FastCGI

1. The Web server creates FastCGI application processes to handle requests. The processes may be created at startup, or created on demand.
2. The FastCGI program initializes itself, and waits for a new connection from the Web server.
3. When a client request comes in, the Web server opens a connection to the FastCGI process. The server sends the CGI environment variable information and standard input over the connection.
4. The FastCGI process sends the standard output and error information back to the server over the same connection.
5. When the FastCGI process closes the connection, the request is complete. The FastCGI process then waits for another connection from the Web server.

Example

```
#include “fcgi_stdio.h" /* fcgi library; put it first*/
#include <stdlib.h>
int count = 0;
void main(void)
{
  /* Response loop. */
  while (FCGI_Accept() >= 0)   {
    printf("Content-type: text/html\r\n",
           "\r\n",
           "<title>FastCGI Hello! (C, fcgi_stdio library)</title>",
           "<h1>FastCGI Hello! (C, fcgi_stdio library)</h1>",
           "Request number %d running on host <i>%s</i>\n",
            count, getenv("SERVER_HOSTNAME"));
    count++;
    if (count > 100) break;
  }
  FCGI_Finish();
}
```

Features

- FastCGI applications can run locally (on the same machine as the Web server) or remotely.
  - For local applications, the server uses a full-duplex pipe to connect to the FastCGI application process.
  - For remote applications, the server uses a TCP connection.
- FastCGI applications can be single-threaded or multi-threaded.
  - For single threaded applications, the Web server maintains a pool of processes (if the application is running locally) to handle client requests. The size of the pool is user configurable.
  - Multi-threaded FastCGI applications may accept multiple connections from the Web server and handle them simultaneously in a single process.
    - For example, Java's built-in multi-threading, garbage collection, synchronization primitives, and platform independence make it a natural implementation language for multi-threaded FastCGI applications.

Code Structure

```
Initialization code
Start of response loop
  body of response loop          // Bascially, the original CGI here
End of response loop
Finalization code
```

- The initialization code is run exactly once, when the application is initialized.
  - Initialization code usually performs time-consuming operations such as
    - opening databases or calculating values for tables or bitmaps.
- The response loop runs continuously, waiting for client requests to arrive.
  - The loop starts with a call to FCGI_Accept, a routine in the FastCGI library.
  - The FCGI_Accept routine blocks program execution until a client requests the FastCGI application.
  - When a client request comes in, FCGI_Accept unblocks, runs one iteration of the response loop body, and then blocks again waiting for another client request.
  - The loop terminates only when the system administrator or the Web server kills the FastCGI application. Suicide is also common when run several times.

FCGI Library

- Use the fcgi_stdio library.
- The fcgi_stdio.h header file contains macros to translate calls to all ISO stdio.h routines into their FastCGI equivalents.
- The fcgi_stdio library provides full binary compatibility between FastCGI applications and CGI applications.
  - You can run the same C binary as either CGI or FastCGI.
- The implementation is in FCGI_Accept.
  - The FCGI_Accept function tests its environment to determine whether the application was invoked as a CGI program or an FastCGI program.
  - If it was invoked as a CGI program, the request loop will satisfy a single client request and then exit, producing CGI behavior.

Implementation Details

- fcgi_stdio.h works by first including stdio.h, then defining macros to replace essentially all of the types and procedures defined in stdio.h.
  - stdio.h defines a few procedures that have nothing to do with FILE *, such as sprintf and sscanf; fcgi_stdio.h doesn't replace these. For instance,
    - FILE becomes FCGI_FILE
    - printf becomes FCGI_printf.
- Some consequences
  - On some platforms the implementation will break if you include stdio.h after including fcgi_stdio.h, because stdio.h often defines macros for functions such as getc and putc.
  - Fortunately, on most platforms stdio.h is protected against multiple includes by lines near the top of the file that look like

    ```
    #ifndef _STDIO_H
    #define _STDIO_H
    ```

Implementation Details

- If your application passes FILE * to functions implemented in libraries for which you have source code, then you'll want to recompile these libraries with fcgi_stdio.h included.
- If your application passes FILE * to functions implemented in libraries for which you do not have source code, then you'll need to include the headers for these libraries before you include fcgi_stdio.h.
  - You can't pass the stdin, stdout, or stderr streams produced by FCGI_Accept to any functions implemented by these libraries.
  - You can pass a stream on a Unix file to a library function by following this pattern:

```
FILE *myStream = fopen(path, "r");
answer = MungeStream(FCGI_ToFile(myStream));
```

Limitations

- The library does not provide FastCGI versions of the functions fscanf and scanf.
  - If you wish to apply fscanf or scanf to stdin of a FastCGI program, the workaround is to read lines or other natural units into memory and then call sscanf.
  - If you wish to apply fscanf to a stream on a Unix file, the workaround is to follow the pattern:

```
FILE *myStream = fopen(path, "r");
count = fscanf(FCGI_ToFile(myStream), format, ...);
```

Memory Leaks

- Memory leaks are seldom a problem in CGI programming
  - Reason: CGI applications rarely run long enough to be concerned with leaks.
- However, memory leaks can become a problem in FastCGI applications, particularly if each call to a popular FastCGI application causes additional memory to leak.
  - Use count!!! E.g., no more than 100 times for each fastcgi.

Perl FCGI API

- accept() – Accepts a new FastCGI connection request, implicitly finishes the current request
- flush() – Flushes output buffers
- finish() – Finishes the current request

Hello World – Perl

```
use FCGI;

$count = 0;

while (FCGI::accept() == 0) {
    print "Content-type: text/html\r\n“,
        "\r\n“,
        "<h1>Hello World</h1>\n“,
        "Request “, ++$count,
        “ from server “, $ENV{'SERVER_NAME'};
}
```

FastCgi Server

- Apache - mod_fastcgi (free) http://www.fastcgi.com/
- Zeus - http://www.zeustech.net/
- Microsoft & Netscape – FastServ plug-in http://www.fastengines.com/