The experimental HTTP Daemon

Avinandan Sengupta

Abstract
This paper presents the rationale, design, implementation, and features of the httpdEx (http daemon Experimental) web server system. The httpdEx web server is a lightweight, multithreaded, and modular system.

Introduction

The httpdEx is an experimental web-server implementation conforming to HTTP 1.1 protocol specification [RFC 2616]. It is CGI 1.1 draft specs compliant and currently implemented on WIN32 platform.

The server has been specifically designed to support web services, which can be developed on top of the HTTP 1.1 protocol. The chief advantages of the system are small memory usage and fast response times. Several features found in standard web server system have not been included in this implementation. Currently the system has support for Directory listing, GET/POST/HEAD/OPTIONS HTTP methods. The architecture is thread-based. The threading mechanism is compile time specifiable - either WIN32 native threads or POSIX threads.

1. Design and architecture

The system has been developed using C. The source code is compiler dependent (presently can be compiled using GNU C compiler on the Linux platform, Borland C++ 5.5 compiler and Microsoft Visual C++ compiler on the WIN32 platform). The runtime libraries provided by certain compiler vendors do not support some of the POSIX functions (e.g. opendir, closedir, readdir, etc.) used. A common denominator port will be available in future. Present implementation of the system is designed for the WIN32 and the Linux platform. The system uses standard windows sockets interface (Berkeley style) for TCP/IP communication.

1.1 System Startup

The server on startup creates a global memory allocator subsystem. This is followed by the creation of the main logger subsystem. The system then creates the configuration subsystem, which reads configuration data from a predefined configuration file. The configuration file is central to the functioning of the system and various parameters for setting up and tuning the server for optimal use. Typical parameters include number of server threads that will be running, port on which the system will listen for HTTP requests, virtual host configuration data, logging levels, etc. Detailed information on the parameters will be presented in a subsequent section.

After the configuration data is loaded, various signal handlers are installed, and synchronization mechanisms are initialized. The MIME data is then loaded into a memory table from the MIME data file as specified in the configuration system. This MIME table is used to map the resources in the HTTP request/response to their MIME types.

1.2 Thread Initialization

The system main thread spawns a thread to initialize the administration subsystem. The thread in turn spawns a predefined number of admin-threads, which the administration requests. Also the main thread spawns another thread to initialize the http subsystem. This thread spawns a predefined number of worker threads. These threads serve the actual HTTP requests. The number of threads which is spawned for serving the administration requests and for serving the HTTP requests depends on the values specified in the configuration file.

1.3 Thread Design

The worker threads wait on the passive socket for incoming HTTP requests. When a request arrives, a specific thread handles it, while the remaining threads wait for serving other incoming requests.

The thread function for the worker threads contains an "accept()" function call that is multithread synchronized (using CRITICALSECTION on the WIN32 platform, and pthread_mutex_t on POSIX compliant systems). The first thread, which calls the function, locks the mutex and waits for incoming requests. When a request arrives, the socket interface returns the client socket to the locking thread. The thread then unlocks the mutex and makes it available to other threads to serve incoming HTTP requests.

1.4 HTTP Request/Response Processing Overview

The thread starts processing the incoming HTTP request from the client socket as follows:

1. A Request object created and initialized.
2. A Response object is created and initialized.
3. The incoming request stream is parsed, and the request object is populated with the parsed values.
4. The request is processed.
5. The response is generated and populated in the response object.
6. The response object is converted into a response stream.
7. The response stream is sent back to the connected client.

2. Administration Interface

The system has been designed to receive administration requests over the network. [The requester is authenticated and the access privilege of the user is checked: this is yet to be implemented]. If the checks are positive, the request sent is parsed. If the request is valid, it is immediately processed. Administration requests for reloading the configuration file, shutting down the system, have been currently implemented. The admin threads process these requests as follows:

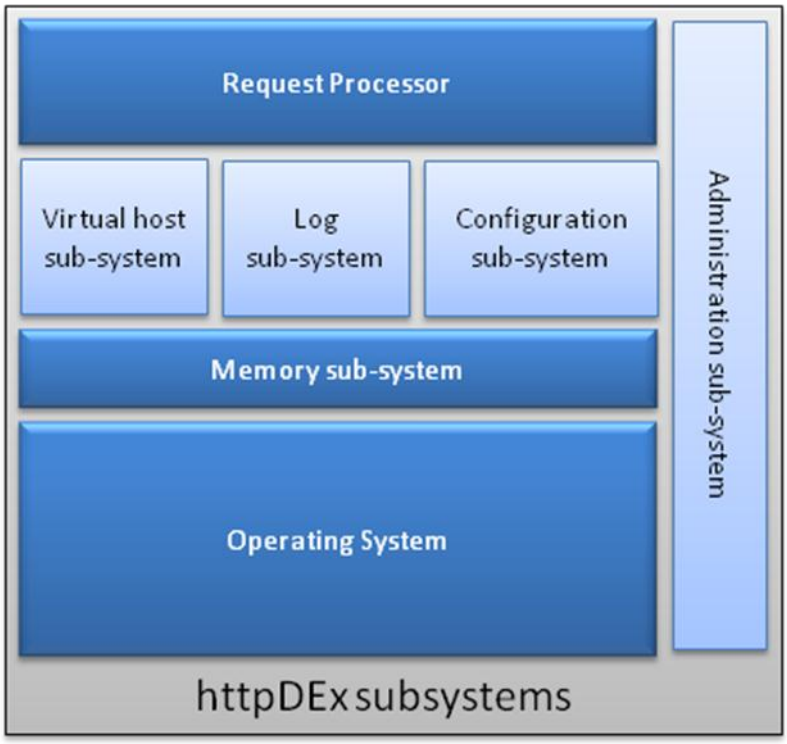
2.1 Administration Request Processing Overview

1. An Administration object is created.
2. The request is parsed and the administration object is populated with the parsed values.
3. The requestor is authenticated, and the privileges of the requestor are checked.
4. If the requestor passed the authentication and has the privileges to issue the command, the system executes the command.
5. The response from the command execution is sent back to the requestor.

Design Note: Administration requests can be sent over the network in an encrypted form. The server decrypts and authenticates the username and password. TLS can be used for transporting the data.

3. Subsystems

The server architecture comprises of a several subsystems. These subsystems have been classified according to the functions they perform.



1. **Request Processor Subsystem:** This subsystem parses the incoming http requests and creates request objects. These request objects are then passed to one of the two request processors – the static request processor for static content requests, or the dynamic request processor for dynamic content requests.

2. **Memory Subsystem:** This subsystem performs memory allocation/de-allocation, memory pooling, garbage collection, and memory fault (leakage/overwrite) detection. This is built on top of the C library malloc/free implementations. All memory operations within the system is done using this library.

3. **Virtual Host Subsystem:** This subsystem provides support for the virtual host mechanism as indicated in HTTP 1.1 protocol. Virtual hosts can be added/deleted/modified dynamically to the server at runtime by changing the details in the configuration file, and reloading the configuration file using the httpC (the httpdEx Controller).

4. **Log Subsystem:** This subsystem provides support for server-wide and virtual host specific logging. The detail of logging is dynamically configurable by setting appropriate log levels in the configuration file.

5. **Configuration Subsystem:** This subsystem maintains the configuration data in memory and is accessed by the server as and when required. The configuration data is loaded from a configuration file into the system.

6. **Administration Subsystem:** This subsystem accepts administration requests from the Controller (httpC) and performs a variety of functions. These functions include reloading the configuration file, shutting down the server, etc.

The system has been designed to ensure fault tolerance and scalability. The number of service threads and the admin threads that will run is configurable through the configuration file.