Web Server Implementation

State Machine

- Server shall:
    - Implement a state machine which consists of the following states:
        - stateIdle: Server is waiting for connections indefinitely. Only Ctrl+C will terminate the server application.
        - stateCreateConnection: Server has received an incoming connection. The incoming connection is verified, and the relevant incoming connection's parameters saved.
        - stateCreatingThread: Server creates a thread to handle requests made by the incoming connection.
    - Support the following events:
        - evtConnection: This event indicates that a TCP connection request has been received and progresses the state machine from stateIdle to stateCreatingConnection.
        - evtConnectionCreated: This event indicates that a successful TCP connection has been established and transitions the state machine from stateCreatingConnection to stateCreatingThread.
        - evtConnectionFailed: This event indicates that a failure occurred while establishing the TCP connection. This event transitions the state machine from stateCreatingConnection to stateIdle.
        - evtThreadCreateSuccess: This event indicates that a thread was successfully created with the appropriate parameters to handle the incoming request. This event transitions the state machine from stateCreatingThread to stateIdle.
        - evtThreadCreateFail: This event indicates that a failure occurred while attempting to create a thread to handle the incoming connection's request. This event transitions the state machine from stateCreatingThread to state Idle.
- Dispatch shall:
    - Implement a state machine which consists of the following states:
        - stateIdle: Upon entering this state, the dispatcher starts a 10 second timer and waits for a valid request to start a new worker thread. If a close request is received from the client, the thread will terminate. If no new requests are received by the incoming connection before the timeout, the thread will exit.
        - stateCreatingThread: The dispatch thread creates a new worker thread to handle a request from the client.
    - Support the following events:
        - evtPendingRequest: Indicates that a valid command was received from the client. Transitions the state machine from stateIdle to stateCreatingThread.
        - evtThreadCreateSuccess: Indicates that a new worker thread has been made to handle the client's request.
        - evtThreadCreateFail: Indicates that an error occurred while attempting to create a worker thread to handle the client's request.
        - evtInvalidRequest: Indicates that a bad command was received from the client.

- Thread shall:
    - Implement a state machine which consists of the following states:
        - stateProcessRequest: Thread begins processing the client's command.
    - Support the following events:
        - evtRequestCompleted: Indicates that the pending request has been completed.
        - evtRequestFailed: Indicates that the thread failed to process the request.

Server Details

- Server should pass a struct to the dispatch thread containing the following information about a connection:
    - File descriptor for incoming connection.
- Server should maintain a list of dispatch threads which are currently running.
- Server shall handle Ctrl+C gracefully, i.e. ensure all dispatch threads terminate before exiting.

Dispatch Thread Details

- Dispatch thread should start a 10 second timer when waiting for new commands from client. If timeout occurs, dispatch thread should close the connection to the client (after all worker threads have completed executing). If a command is received from the client, restart the timer (even if the command was invalid).
- Dispatch thread should maintain a list of all worker threads it has created.
- If the server signals to terminate (from a Ctrl+C command), ensure all worker threads have completed, and close all open TCP connections.
- Dispatch thread should parse the incoming client command. If it is a GET/POST command, create a new worker thread and pass to it the appropriate parameters.

Worker Thread Details

- Every worker thread will only handle a single command from the client. This includes, transmitting the requested file or sending the correct response.

Necessary Files

The following files shall be created:

- state_machine.c/.h
    - All code related to the state machines.
- file.c/.h
    - Code related to file handling.
- threading.c/.h
    - Code related to multithreading.
- html.c/.h
    - Code related to html requests (parsing, packing, structs).
- socket.c/.h
    - Code related to socket handling.

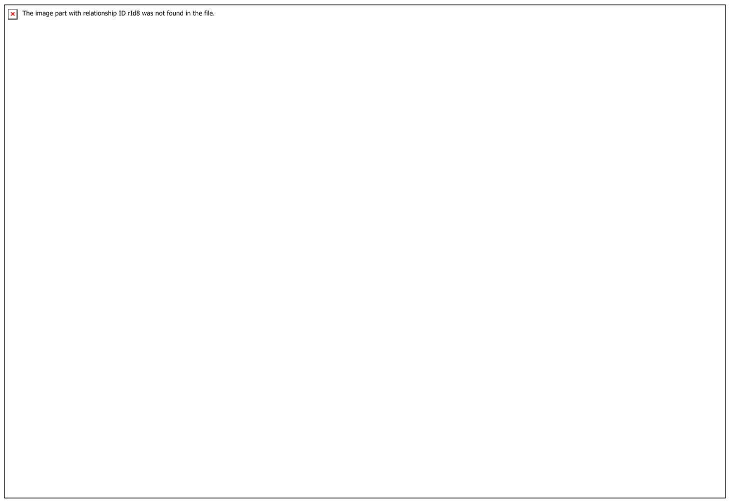
HTTP Implementation

References:

- [HTTP Request Message format well explained]
- [HTTP - Requests]
- [HTTP - Responses]

The webserver shall:

- Support HTTP/1.0 and HTTP/1.1
- Implement HTTP packet processing such as:
  - Parsing Fields:
    - Request Line
    - Header Fields
    - Empty Line
    - Message Body
  - Generating HTTP response packets:
    - Response Line
    - Header Fields
    - Empty Line
    - Message Body


The image part with relationship ID rId8 was not found in the file.

HTTP Requests (In Detail)

Request-Line

- [FORMAT] Request-Line = Method SP Request-URI SP HTTP-Version CLRF
- Request Method
  - SUPPORTED BY HTTP/1.1
    - GET
      - Used to **retrieve** information from the given server using a given URI.
    - HEAD
      - Same as GET but transfers the status line and the header section only.
    - POST
      - Used to send data to the server.
    - PUT
      - Replaces all the current representations of the target resource with the uploaded content.
    - DELETE
      - Removes all the current representations of the target resource given by the URI.
    - CONNECT
      - Establishes a tunnel to the server identified by a given URI.
    - OPTIONS
      - Describes the communication options for the target resource.
    - TRACE

            - Performs a message loop back test along the path to the target resource.
    - SUPPORTED BY HTTP/1.0
        - GET
        - HEAD
        - POST
- Request URI
    - [FORMAT] Request-URI = "*" | absoluteURI | abs_path | authority
        - The asterisk * is used when an HTTP request does not apply to a particular resource, but to the server itself, and is only allowed when the method used does not necessarily apply to a source.
        - The **absoluteURI** is used when an HTTP request is being made to a proxy. The proxy is requested to forward the request or service from a valid cache, and return the response.
        - The most common form of a Request-URI is that used to identify a resource on an origin server or gateway. For example, a client wishing to retrieve a resource directly from the rogin server would create a TCP connection to port 80 of the host www.w3.org and send the following lines:
            - GET /pub/WWW/TheProject.html HTTP/1.1
            - Host: www.w3.org

          Note that the absolute path CANNOT BE EMPTY. If none is present in the original URI, it MUST be given as "/" (the server root).

Request Header Fields

- Request header fields allow the client to pass additional information about the request, and about the client itself, to the server. These fields act as request modifiers.

Request Message Field

- Used to send actual data to the server (like during a POST request).

HTTP Responses (In Detail)

Message Status-Line

- Consists of the protocol version followed by a numeric status code and its associated textual phrase. The elements are separated by space SP characters.
- [FORMAT] Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
- HTTP Version
    - A server supporting HTTP version 1.x will return the following version information:
        - HTTP-Version = HTTP/1.x
- Status Code
    - A 3-digit integer where first digit of the Status-Code defines the class of response and the last two digits do no have any categorization role. 5 possible values for the first digit.
        - 1xx: Informational

        - 2xx: Success
        - 3xx: Redirection
        - 4xx: Client Error
        - 5xx: Server Error

Response Header Fields

- Allow the server to pass additional information about the response which cannot be placed in the Status-Line. These header fields give information about the server and about further access to the resource identified by the Request-URI.

Response Message Body

- Contains the body of what is being sent (i.e. a file/image/etc)

Notes

2/22/22

Current Progress:

- The web server can spawn a dispatcher thread whenever a new connection is made. The spawned dispatcher thread in turn listens for a single message from the client, echo's the message to the terminal, closes the connection, then terminates.

Next Up:

- Need to **parse the HTML commands** in the dispatcher thread to validate if the command received is any good. If it is a valid command, the **dispatcher thread will create a worker thread to handle the requested job** (GET/POST commands).
- The spawned worker thread will handle sending the appropriate response to the client. This could include opening a file, reading from the file, and transmitting the file over the TCP connection.
- ~~Once the worker thread has completed, it will simply terminate. Every thread generated in the application will be detached so we won't have to wait for the threads to terminate in the main application/dispatch thread.~~

Future:

- ~~Will need to implement the 10 second timeout in the dispatcher thread when waiting for more messages from the client.~~
- Will also need to implement graceful termination from entering application whenever entering "ctrl+C" from the command line.

Notes:

- Since we're running our code on elra-01.cs.colorado.edu, if we want to send HTTP commands from the web browser, we need to VPN into the school network if we're off campus. If we're on campus, should be fine. Just remember to use the server's actual IP address (198.59.7.11).

2/23/22

Current Progress:

- Web server can currently spawn a unique dispatcher thread for each unique connection. Each dispatcher thread in turn spawns a new worker thread whenever the remote client sends a command/request.

Next Up:

- **Need to parse HTTP request in the dispatcher**.
    - If the request is valid, the dispatcher will spawn a thread and pass a variable which contains the appropriate parameters that the worker thread needs to complete its task.
    - If the request contains a keep-alive parameter, the dispatcher should create a timer for subsequent listens, otherwise terminate immediately.
    - If the request is invalid, the dispatcher should respond appropriately.

Future:

- Handle graceful termination of application when Ctrl+C is detected.

2/27/22

Current Progress:

- Currently working on HTTP packet parsing in the dispatcher. Able to parse request line. Currently parsing the request headers. Payload should be as simple as copying.

Next up:

- Once request header is complete, and the request is considered valid, need to memcpy the payload (if there is one), pass that to the p_results struct, and create the worker thread to handle the request.

2/28/22

Current Progress:

- Web server can currently spawn a unique dispatcher thread for each unique connection. Each dispatcher thread in turn spawns a new worker thread whenever the remote client sends a command/request. The worker thread now has access to the necessary parameters it needs to complete a job.

Next Up:

- **Create file opening/reading method to get file contents (using request URI).**
- **Create HTTP response packet generating method.**
    - Method should create a buffer and fill it with the necessary contents such as:
        - Start-line
        - Header field
        - Empty line
        - Body
    - Ex
    -
      The image part with relationship ID rId8 was not found in the file.

Future:

- Handle graceful termination of application when Ctrl+C is detected.
- Properly handle timeouts when connection is not set to keep-alive.
- Explore the segfault condition that occurs whenever a client disconnects.

3/4/22

Current Progress:

- Web server can kind of send response packets to the client (chrome)

Next Up:

- We are experiencing seg-faults and memory errors. Need to resolve.
- Some packets are not getting sent, not sure why this is happening.
- Figure out why we are getting memory issues.

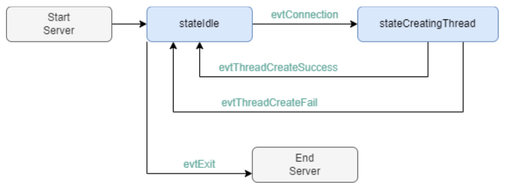
Notes:

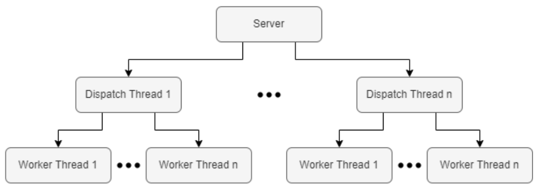
- Run the server using: valgrind -s --leak-check=full --show-leak-kinds=all --track-origins=yes ./web_server 60002 > results.txt

3/8/22

It’s done. POST and GET commands are working now. Server does not segfault anymore upon frequent client requests. It’s over. Ah.

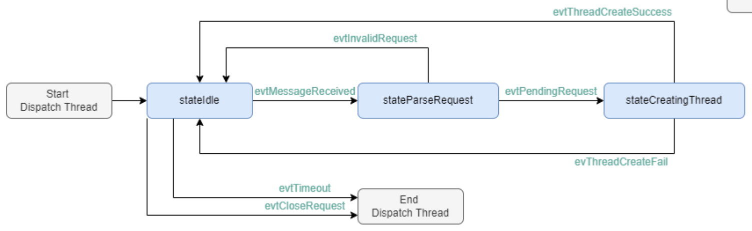
Appendix

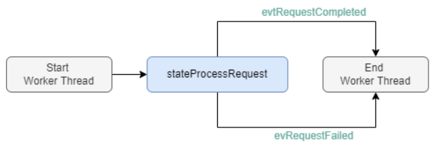




Basic Procedure

For every unique incomming client, create a Dispatch thread to handle listening for client requests. For each client request that is received by the dispatch thread, a new worker thread will be created to handle the request.


evtThreadCreateSuccess
evtInvalidRequest
Start
Dispatch Thread
stateIdle
evtMessageReceived
stateParseRequest
evtPendingRequest
stateCreatingThread
evThreadCreateFail
evtTimeout
evtCloseRequest
End
Dispatch Thread


evtRequestCompleted
Start
Worker Thread
stateProcessRequest
End
Worker Thread
evRequestFailed