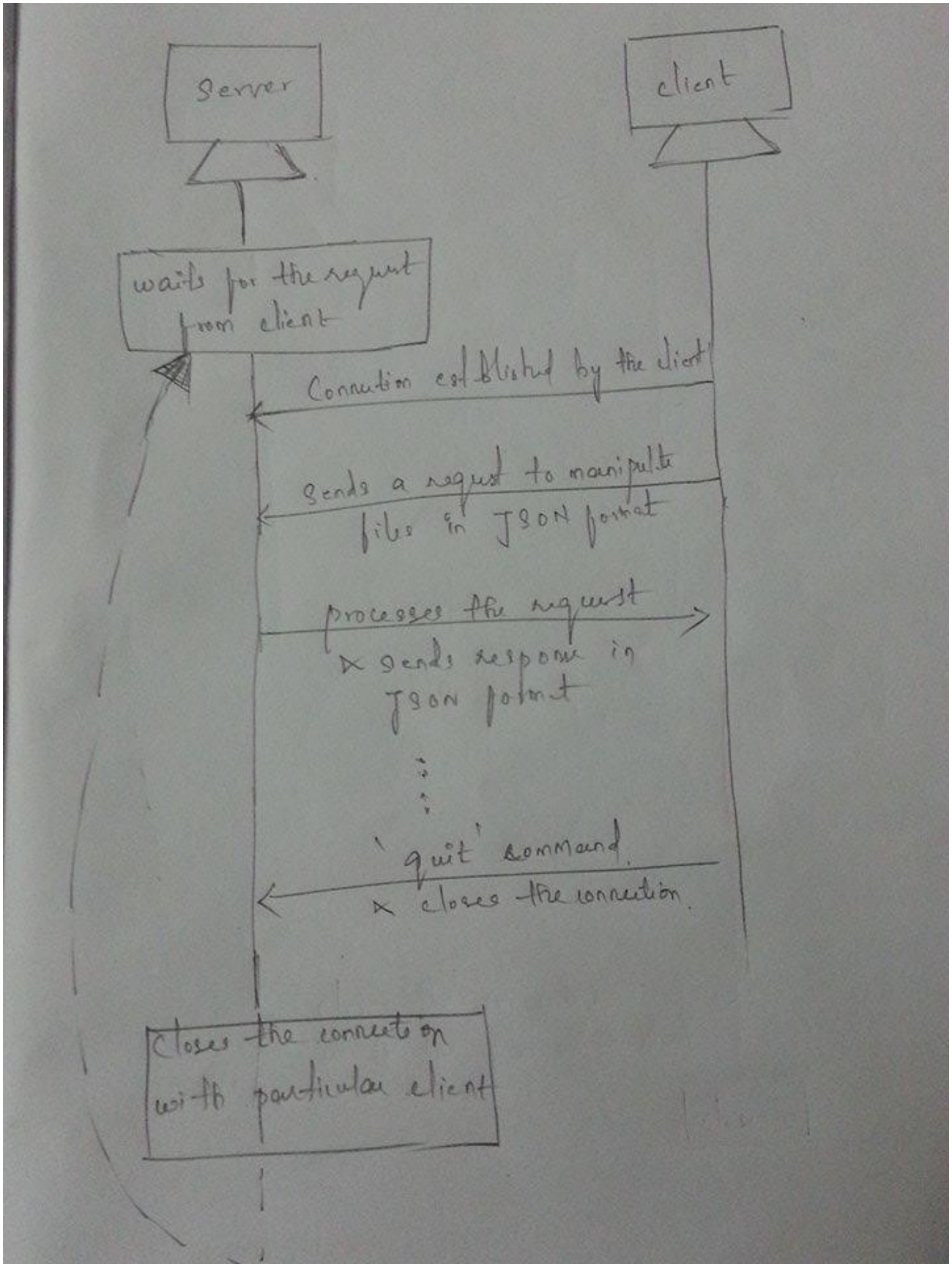
Design and Implementation

Design



Implementation:

Assumptions:

- TCP is used to establish connection to ensure reliability
- Client and server runs on the same machine (both uses 'localhost' for connection establishment)
- Client always makes request in JSON format except "quit"(signal to terminate connection) command and server responds back in JSON format.
- No authentication mechanism is used when a client establishes connection with server. To improve security, an authentication mechanism can be used during connection establishment between client and server.
- Client closes the connection by issuing "quit" to the server. Server closes the connection with the client after receiving "quit"
- Both the server and client receives the data in small chunks(1024 bytes). Server identifies the end of request by examining the string "EOReq" from client. Client identifies the end of response by examining the string "EOResp" from server
- Client can wrap multiple requests into single request. And the server unwraps each request, processes them individually and sends the response in JSON format(for request separately). Once the server has serviced all the requests, server sends a special UUID("bfd99f05-befb-4b9d-8fd1-e5f517b30685") to client. On receiving "bfd99f05-befb-4b9d-8fd1-e5f517b30685" from server, the client ensures that the server is ready to process the next request and issues the consecutive request. This continues until client issues "quit". Another approach : Here the UUID can be generated randomly for each request, and server can associate this UUID in each response

  example request with all operations combined :{"read":[{"filename":"test.txt"}, {"filename":"test.txt", "offset":5}],"update":[{"filename":"test.txt", "data": "Works fine"}], "delete":[{"filename":"test.txt", "end":1,"num_chars":4}]}
- Server processes only valid JSON requests. If the request is not a properly formatted JSON, then the server issues "Invalid request" error
- Whenever the request has attribute "end":1, it is interpreted as the following
  in read request : data will be read from end
  in update request : data will be appended in the end
  in delete request : data will be deleted from the end

Server Operations:

For all the server operations 'filename' is mandatory without filename, the server cannot process the request.

Read:

1. read(filename, offset, num_chars) : Reads data(num_chars - number of characters) from specified filename, from specified offset.
   a. if num_chars = 0, then the entire will be read

b. if num_chars > total characters in file, the entire file will be read

2. readE(filename,num_chars) : Invoked when request has “end”:1 attribute. Reads from the end but this operation is not accessible for the client directly, based on the client request the server redirects the request to this function. ‘num_chars’ field is mandatory to read data from end
   example : {"read":[{"filename":"test.txt", "num_chars":23, "end":1}]}

Update:

1. update(filename, data, start_offset, end_offset): Updates the given data in specified file from the specified start_offset. ‘data’ field is mandatory for update request
   a. if both start and end_offset is specified, it replaces the data from start to end_offset with the user specified data.
   b. if only end_offset is specified, it replaces the data from 0 to end_offset with the specified data.
   c. if none of start and end_offset is specified, the data will be prepended to the file.
   d. if start_offset>end_offset, server issues “Invalid offset” error

2. updateE(filename, data): Invoked when request has “end”:1 attribute. Appends the given data to the file content. This function is not directly accessible for the client, the server redirects based on the request parameters. ‘data’ field is mandatory
   example : {"update":[{"filename":"test.txt","data":"\nLast line\n", "end":1}]}

Delete:

1. delete(filename, start_offset, end_offset, num_chars): Deletes the specified no.of characters from file from the specified start_offset
   a. if start and end offset is specified, then ‘num_chars’ field will be ignored. data between the start and end_offset will be deleted
   b. if start and num_chars specified, deletes specified num_characters from start_offset
   c. if start_offset>total character in file, then no data will be deleted from file
   d. if start_offset>end_offset, server issues “Invalid offset” error
2. deleteE(filename, num_chars): Invoked when request has “end”:1 attribute. This function deletes the data (specified num_chars) from end of the file.’num_chars’ field is mandatory
   example : {"delete":[{"filename":"test.txt", "end":1, "num_chars":4}]}

Client Operations:

Client can issue any combination of read, update and delete requests. “quit” to terminate the connection
Client Request:

1. {"update":[{"filename":"test.txt","data":"first data to get updated", "end":1},
   {"filename":"test.txt","data":"first data to get updated", "start_offset":4}]}
2. {"delete":[{"filename":"test.txt", "num_chars":4}]}
3. {"read":[{"filename":"test.txt", "offset":3, "num_chars":4}]}
4. {"read":[{"filename":"test.txt"}], "update":[{"filename":"test.txt", "data":"Works fine"}]}

Server Response:
Attributes : Request (Actual request sent from client), Response(Server’s response for the request)
"{\"Request\": \"read([{'filename': 'test.txt'}, {'offset': 5, 'filename': 'test.txt'}])\", \"Response\": \"Works fineworks fin\"}"

"{\"Request\": \"read([{'filename': 'test.txt'}, {'offset': 5, 'filename': 'test.txt'}])\", \"Response\": \" fineworks fin\"}"

"{\"Request\": \"update([{'data': 'Works fine', 'filename': 'test.txt'}])\", \"Response\": \"File updated successfully\"}"

"{\"Request\": \"delete([{'num_chars': 4, 'end': 1, 'filename': 'test.txt'}])\", \"Response\": \"Requested file contents deleted successfully\"}"

Algorithm Definition:

server {

1. creates TCP socket, binds its address and port
2. listens for the incoming connections
3. on request from client, establishes connection with client
4. receives request and validates the request
5. on successful validation, processes the request and sends response to client over TCP
6. closes the connection once it receives “quit” from client

}

client{

1. create TCP and connects to server by using server address and port
2. gets request from user and sends it to the server over TCP
3. after sending each request, waits for the response from server.
4. after receiving UUID from server, client sends the next consecutive request
5. user issues “quit” to terminate the connection and closes its socket

}