Design Document

By: Team 33
Brian Pham – bnp200000
Nick Tollinger – ngt190001
Akshaan Singh – axs210087

Date: 03/21/2025

Table of Contents

Server Documentation

The server will act as the centralized calculator hub that any client can connect to, provided that the client runs on the same IP address and port as the server. When the server starts, it will first create a list of **ConnectedClient** objects representing the clients currently connected to that server. It will also create an **Infix** object that will be used to handle the parsing and calculation of mathematical equations.

Once the server has started running, it will create a **ServerSocket** object that it can connect to using the provided PORT number (default 5000). Once a socket is created, the server will run a method called ***initConnection()*** to handle incoming client connections.

initConnection() → void

The method ***initConnection()*** is responsible for handling incoming client connections. Each time a client attempts to connect to the server, the server socket will attempt to accept the incoming client; it will terminate prematurely if the client fails to connect to the socket.

Once the server socket accepts the incoming client's request to connect to the server, it will create a **DataInputStream** object to read in the client's name. If a name is not given, an error statement will be outputted and the connection will be closed.

For the server to handle multiple connections from different clients, multithreading is utilized to handle the multiple client connections. Each time a thread is started, a **ConnectedClient** object is created that takes in the socket object (used to connect the client to the server socket) and an index number (for logging purposes). After that, the object is added to the list of **ConnectedClient** objects defined in the **Server** class.

Once the object has been created and added to the list, the ***handleClientRequest()*** method is called to handle any requests that the client will send to the server.

Once the client has disconnected from the server, the following information will be logged by the server using the ***logClient()*** method:

- The client's name and index number
- The time a connection was established
- The duration of the client's connection to the server
- The requests sent from the client to the server

- The time a connection was closed

The server is synchronized to ensure that the threads will not be deadlocked when it is writing to a log file.

handleClientRequest(ConnectedClient client) → void

The method ***handleClientRequest()*** is used to handle incoming math equations from the client to the server. While the client is currently connected to the server, the Infix object will execute its ***evaluate(eq)*** method to parse the client's incoming math equation and store the result in the ***sendResponse(value)*** method defined in the **ConnectedClient** object; it will also log the equation and the result in the ***logRequest(eq, value)*** method also defined in the **ConnectedClient** object. Once the client disconnected from the server, a ***setDisconnectTime(time)*** method – also defined in **ConnectedClient** – is called to set the current date and time of the disconnection

logClient() → void

The ***logClient()*** method is used to log any client that has connected to the server. What it does is that it will create a ***Log/*** directory to store all of the log (.txt) files used for the session that the server is running.

For each client in the list of **ConnectedClient** objects, the **FileWriter** object is used to log the following information about the client:

- The client's name and index number
- The time a connection was established
- The duration of the client's connection to the server
- The requests sent from the client to the server
- The time a connection was closed

Once the file has been fully written, the stream is flushed to ensure that the buffered output bytes are written out.

Client Documentation

The client represents the user who will attempt to connect to the server and feed it math equations for it to solve. Since the calculator application utilizes TCP, it must wait for the server to start running before it can connect.

The client will attempt to connect to the server's socket by supplying the IP address and port to which we want to connect. For our application, we will assume that the server is running on the same IP address as the client and using port 5000. Thus, removing the need for the user to supply the IP address and port number if the application only needs to run on one machine with multiple terminals. Once a socket connection is established, the input and output streams will be set up to handle communication between the client and server.

- The **BufferedReader** object is used to handle client-side input
- The **DataOutputStream** object is used to send client messages to the server
- The **DataInputStream** object is used to handle server responses

The client will abruptly end if it fails to connect to the host or the I/O streams.

Once the socket connection and I/O streams have been established, the client will send its name to the server using the output stream for server-side logging. After that, the ***writeToServer()*** method is called to handle sending math equations to the server. The ***close()*** method is called to close the socket and I/O streams.

writeToServer() → void

The ***writeToServer()*** method handles sending messages to the server and receiving responses. While the client is connected to the server, it will be prompted to enter a math equation ('#' to close the connection). The **DataOutputStream** object is used to write the client's input to the server. The **DataInputStream** object is used to receive the server's response; the client terminal will output the response.

close() → void

The ***close()*** method will close the socket connection and all I/O streams.

getLocalAddress() → void

The ***getLocalAddress()*** method will get the IPv4 address of the client, since we assume that the network application will only run on one machine with one server terminal and multiple client terminals. If an address fails to be retrieved, it will simply use the localhost address $127.0.0.1$ as the default address to connect to the server.

ConnectedClient (Middleware) Documentation

The **ConnectedClient** is used to handle clients that are connected to the server; its primary function is to act as the middleware between the centralized server and the multiple clients.

The **ConnectedClient** object expects a socket, ID number, and the name of the client connected to the server. The **DataInputStream** and **DataOutputStream** are set up using the client's input and output streams.

read() → String

The ***read()*** method is responsible for reading messages from the client using the input. It will be used by the centralized **Server** to read in mathematical expressions.

sendResponse() → void

The ***sendResponse()*** method is responsible for retrieving the result from the server and sending it back to the client's output stream.

close() → void

The ***close()*** method is responsible for closing the client's socket connection to the server and its I/O streams.

getName() → String

The ***getName()*** method returns the client's name

getId() → int

The ***getId()*** method returns the client's ID number

getStartTime() → LocalDateTime

The ***getStartTime()*** method returns the date and time the client connected to the server.

getDisconnectTime() → LocalDateTime

The ***getDisconnectTime()*** method returns the date and time the client disconnected from the server

setDisconnectTime(LocalDateTime disconnectTime) → void

The ***setDisconnectTime()*** method sets the client's disconnection time to **disconnectTime**.

logRequest(String eq, double result) → void

The ***logRequest()*** method handles logging every message that the client has sent to the server and the response it receives for each of those messages.

getRequestLog() → List<String>

The ***getRequestLog()*** method returns the logging list of a client's interaction with the server, such as messages and responses.

Infix (Backend) Documentation

The **Infix** is the calculator backend. It is responsible for handling mathematical expressions – parsing them and evaluating them. When a client sends a math expression for the server to solve, the server will call the **Infix** backend to parse the expression and perform the calculation.

The **Infix** utilizes an **operand** stack to handle the operands (2, -3, 5.4) and an **operator** stack to handle the operators (+, -, *, /). It will also use a **bitCheck** flag to check if the equation was parsed correctly; it defaults to 1 for success.

evaluate(String expr) → double

The ***evaluate()*** method is responsible for performing the calculation on the input expression. What it does is parse the expression – it sanitizes the expression and tokenizes it into a list of tokens, i.e., "2 + 3(4 - 9)" will become ["2", "+", "3", "*", "(", "4", "-", "9"].

The **bitCheck** flag can change depending on the result of the tokenization process; if the tokenization fails for any reason, the method will abruptly end if **bitCheck** is 0 and return a **Double.NaN** value to indicate that the expression failed to parse, likely due to a client-side error, such as input format.

Otherwise, the method will evaluate each token in the list of tokens.

- If the token is an operand, it will be converted into a **Double** value and pushed onto the **operand** stack.
- If the token is an open parenthesis '(', it will be pushed into the **operator** stack to signify the beginning of a sub-expression.
- If the token is a closing parenthesis ')', it will do the following:
    - While the operator stack is not empty and the topmost item is not an open parenthesis '(', it will call the ***operate()*** to operate on the current tokens in the **operand** and **operator** stacks. Once the loop is finished, it will pop the latest item in the **operator** stack – the open parenthesis '(' to signify the end of a sub-expression.
- If the token is an operator, it will do the following:
    - While the **operator** stack is not empty and the token's operator precedence is less than or equal to the topmost operator in the **operator** stack, it will call the ***operate()*** to operate on the current tokens in the **operand** and

operator stacks. Once that loop is finished, it will append the token onto the **operator** stack.

Once all the tokens have been looped through, it will finalize any remaining tokens in the **operand** and **operator** stacks by continuously calling the ***operate()*** method until the **operator** stack is empty. After that, it will pop out the topmost item in the operand – the final result.

isOperator(char c) → boolean

The ***isOperator()*** method checks if **c** is any of the following math operators: *, /, +, -, %, or ^.

isOperand(String op) → boolean

The ***isOperand()*** method checks if **op** is a valid operand, i.e., 2, -3, 5.4, using regex matching.

precendence(char c) → int

The ***precedence()*** method returns the precedence value of **c** based on PEMDAS ordering.

- If **c** is the exponent (^) operator, the return value is 3
- If **c** is the multiplication (*), division (/), or modulo (%) operator, the return value is 2
- If **c** is the addition (+) or subtraction (-) operator, the return value is 1
- Otherwise, the return value is -1.

operate() → void

The ***operate()*** method is responsible for performing a math operation on the two most recent items in the operand stack using whatever operator is currently on top of the **operator** stack. After that, the result is pushed onto the **operand** stack for further usage by the ***evaluate()*** method.

parseExpression(String expr) → List<String>

The ***parseExpression()*** method is responsible for sanitizing the input expression and splitting it into a list of tokens. It will also keep track of the number of operators, operands, open parentheses, and closing parentheses found in the sanitized expression to ensure that the expression is valid.

While the sanitized expression can be looped through, the method will do the following, based on the current character **curr**:

- If **curr** is an operator, increment the **numOperators** value and append it to the tokens list.
- If **curr** is an open parenthesis ‘(‘, increment the **numOpenParen** value. If the tokens list is not empty and the last item in that list is an operand, increment the **numOperators** value and append the ‘*’ to the tokens list. After that, append **curr** to the tokens list.
- If **curr** is a closing parenthesis ‘)’, increment the **numOpenParen** value and append it to the tokens list.
- If **curr** is an operand, converting it will be a lengthy process since **curr** is only a single character at the moment.
    - Increment the **numOperands** value. After that, loop under the following conditions for the next index $i + 1$:
        - It is less than the length of the sanitized expression string AND
        - It is not an operator AND
        - It is not a ‘(‘ AND
        - It is not a ‘)’

      During that loop, the **curr** value will be appended with the character value at the index $i + 1$ and increment the i value.

  Once the full operand value has been built, convert it into a double value. Otherwise, reset the **numOperands** value back to 0 and throw an error. After that, it will handle special cases, such as negative numbers or expressions matching A(B).

Once the entire token has been built, set the **bitCheck** value to 0 if any of the following condition occurs separately:

- **numOperands** is 0; no operand detected
- **numOperators** is 0; no operator detected
- **numOperators** ≥ **numOperands**; the number of operators is greater than or equal to the number of operands
- **numOpenParen** ≠ **numClosedParen**; the expression is unbalanced – the number of opening parentheses does not match the number of closing parentheses

The final step is to simply return the finalized list of tokens for the ***parseExpression(expr)*** method to use.