Server side

The server project includes features such as handling client requests, managing child processes, and logging. Let's go through the different components and design choices made in the code.

Overall Structure

The server project consists of multiple source files, including `queue.h`, `file_handler.h`, and `log.h`. These files provide necessary data structures and functions used in the server implementation. The main server code is located in the `main` function of the main source file.

Server Struct

The server struct, `server`, is defined to hold various information and state related to the server. Here are the key fields of the `server` struct:

- `max_client`: Maximum simultaneous client capacity.
- `directory`: The directory on which the server runs.
- `connection_fd`: File Descriptor (Fifo) for incoming requests.
- `children`: Array to store the child process PIDs.
- `request_queue`: A queue data structure to hold incoming requests.
- `available_child_size`: Size of available children.
- `connection_path`: Path for the server's FIFO file.

Signal Handling

The server project includes signal handling for `SIGINT` and `SIGCHLD`. The `sigintHandler` function is responsible for handling the `SIGINT` signal, which is used to terminate the server. When the signal is received, it checks whether the process is the parent process or a child process and takes the necessary actions accordingly.

The `handle_SIGCHLD` function is the signal handler for `SIGCHLD`, which is triggered when a child process terminates. It updates the server's child process list, marks the child process as finished, and processes the request queue.

Initialization

The `init_server` function is responsible for initializing the server struct and performing the necessary setup operations. It takes the maximum number of clients and the server directory as input parameters. In this function, a directory is created for the server, a FIFO file is created for incoming requests, and the FIFO file is opened for reading.

Request Handling

The server handles client requests in the main loop of the program. The `get_request` function reads a request from the server's FIFO file. The `handle_request` function processes the received request. If there are available child processes, the request is added to the request queue. If there are no available child processes and the request is a "tryConnect" request, a response of "wait" is sent to the client.

Child Process Handling

The server manages child processes using an array of child process PIDs. When a child process terminates, the `handle_SIGCHLD` function updates the child process list and marks the child process as finished. The `check_children` function is responsible for checking the status of child processes and making them available if they have terminated.

The `process_queue` function is called to process the request queue. If there are available child processes, a request is dequeued from the queue, and a child process is forked to handle the request. The child process runs the `client_handler` function, which handles the request-specific operations and interacts with the client.

Logging

The server project includes logging functionality using the `log.h` file. The `logMessage` function is used to log messages to a log file. The `client_handler` function opens a log file specific to the client and logs each request before processing it.

Server side provides a foundation for handling client requests, managing child processes, and logging. The design ideas presented in this report offer a starting point for further development and customization based on specific requirements and additional functionality needs.

File Operation Handling

This part is a file handler for a server that handles file operations. Let's go through the code and explain its functionality and design ideas:

0. The `handle_client` function is the main handler for each connected client. It takes the client file descriptor as input and performs the necessary file operations based on the received commands.

1. The `split_cmd` function is defined to split a command into separate arguments. It takes a command string and an array of arguments, splits the command based on spaces, and returns the number of arguments.

2. The `help` function prints the available commands if no command is specified. It provides information about the available commands such as "help," "list," "readF," "writeT," "upload," "download," "quit," and "killServer." However, the implementation of the `help` function is incomplete and marked with a TODO comment.

3. The `path_merge` function is used to merge a directory path and a file name. It concatenates the directory and file name with a slash in between and returns the merged path.

4. The `list` function takes a directory path as input and lists the files in that directory using the `readdir` function. It iterates over the files and prints their names using `printf`. This function is responsible for listing the files in a given directory.

5. The `seek_to_line` function takes a file descriptor and a line number as input. It reads the file character by character until it reaches the specified line number. It returns the offset after the newline character of the specified line. This function is used to seek to a specific line in a file.

6. The `print_until_newline` function takes a file descriptor as input and reads the file character by character until it reaches a newline character. It prints the characters using `putchar` and stops at the newline. This function is used to print the content of a file until a newline character is encountered.

7. The `readF` function takes a file path and a line number as input. It opens the file using the `open` function with read-only permissions. If the file cannot be opened, an error message is printed. If the line number is 0, the entire file is read and printed using a buffer. If the line number is not 1, the function seeks to the specified line using the `seek_to_line` function and prints the content until the next newline using the `print_until_newline` function.

8. The `writeT` function takes a file path, a line number, and text as input. If the line number is 0, the function opens the file in append mode and writes the text at the end of the file using `fprintf`. If the file does not exist, a new file is created with the given text. If the line number is not 0, the function opens the original file for reading and a temporary file for writing. It reads the original file line by line, and if the current line matches the specified line number, it writes the new text to the temporary file. Otherwise, it writes the original line to the temporary file. After processing all the lines, the original file is closed, and the temporary file is renamed to the original file name using the `rename` function. If the line number is greater than the number of lines in the original file, the new text is appended at the end of the file using the `fprintf` function.

9. The `upload` function is responsible for receiving a file from the client and saving it on the server. It takes a file path as input and opens the file in write mode using the `fopen` function. If the file cannot be opened, an error message is printed. Otherwise, it reads the file contents sent by the client using the `read` function and writes them to the file using the `fwrite` function. Finally, it closes the file using the `fclose` function.

10. The `download` function is responsible for sending a file from the server to the client. It takes a file path as input and opens the file in read mode using the `fopen` function. If the file cannot be opened, an error message is printed. Otherwise, it reads the file contents using the `fread` function and sends them to the client using the `write` function.

11. The `quit` function is used to gracefully terminate the server. It closes the client fifo, cleans up any remaining temporary files, and exits the program.

12. The `killServer` function is used to forcefully terminate the server. It calls the `quit` function to perform the cleanup and then sends the SIGINT signal to itself using the `kill` function.

Overall, this part represents a file handler that handles file operations such as listing files in a directory, reading a file, writing to a file, uploading a file from the client, and downloading a file to the client.

Client Side

The client-side code provided is a part of a client-server application. It is responsible for establishing a connection with the server, sending commands, receiving responses, and performing file upload and download operations. This report provides a detailed analysis of the client code, including design ideas and implementation details.

Overall Structure
The client code consists of several functions and data structures that facilitate communication with the server. The main function serves as the entry point for the client program. It initializes variables, establishes a connection with the server, handles user input, and processes server responses. The code also includes signal handling to capture the SIGINT signal for graceful termination.

Data Structures
The client uses a structure called \`client\` to hold the file descriptors for input and output streams with the server. This structure enables efficient communication between the client and server processes.

Connection Establishment
The \`connect\` function is responsible for connecting the client to the server. It takes the server's process ID and a try parameter as input. The function first creates a response FIFO path unique to the client process. It then opens this FIFO to receive responses from the server. Next, it constructs a connection request message and sends it to the server through the main server FIFO. If the server is busy, indicated by the response "wait," the function waits until the server is available. Once the server is ready, it returns the child server's process ID, which is used to establish a bidirectional connection between the client and child server.

Command Processing
The client reads commands from the user and sends them to the server for execution. The user can enter various commands, including "upload" and "download," which trigger file upload and download operations, respectively. The "quit" command is used to exit the client program. The client sends the command to the server using the output file descriptor. If an "upload" command is encountered, the client calls the \`upload_file\` function to upload the specified file to the server. For a "download" command, the client calls the \`download_file\` function to download the requested file from the server.

File Upload and Download
The `upload_file` function handles the file upload process. It opens the specified file for reading and sends its content to the server using the output file descriptor. The client reads the file in chunks and writes them to the output file descriptor until the entire file is uploaded.

The `download_file` function manages the file download process. It checks if the file already exists on the client-side and creates a unique filename if necessary. The client reads data from the input file descriptor and writes it to the newly created file until there is no more data to read.

Signal Handling
The client code includes a signal handler function `sigintHandler` to capture the SIGINT signal (Ctrl+C). When the SIGINT signal is received, the handler sends a kill request to the server process using the `kill` system call. This allows the server to gracefully terminate and clean up any resources before the client exits.

Conclusion
The client code demonstrates a robust implementation for establishing a connection with the server, sending commands, and processing server responses. The code incorporates file upload and download functionality, providing a complete client-side solution. The signal handling mechanism ensures proper termination of the server process when the client receives the SIGINT signal. Overall, the code reflects good design principles and offers a solid foundation for building a reliable client-server application.

Tests
All tests screen image files are included in tests directory. All commands tested.