**Client-Server Architecture**

**Introduction:**

This project implements a client-server architecture using c/c++ programming language and Unix APIs. The architecture has three entities, the server, one or more clients and their respectful client handlers. The internal workings and used techniques will be thoroughly discussed in this report.

**Server:**

The server is the center and the most integral part of this architecture. Its responsible to connect the clients and process their commands and requirements. Server is implemented with a sockets programming architecture that makes it capable of accepting client connections from both the same and any other physical machine that is connected to the network. For now, it is designed to accept connections only from the host’s private network, different configurations such as port forwarding need to be implemented in order to accept from public networks.

**Client Handler:**

Another important entity is the client handler which is responsible to handle and process the client requests. As the server is not capable of both accepting of client connections and processing the active client commands simultaneously, it is important to dedicate a separate process for each client. Client Handler comes with various commands such as different Arithmetic operations, run, kill etc. that the clients can use to get things done!!

**Client:**

Another important is your client that needs to connect with a server to start processing instructions. The client acts as a frontend to receive the instruction from the user and then the client handler (backend dedicated server) processes and the output is displayed on the client’s end.

**Multi-Threading:**

In server, multi-threading was implemented so that the server can accept incoming connections and handle user commands simultaneously. The commands like print and list can be used on the server end to send messages to active clients or receive process lists of active clients.

In case of the client, threading was implemented so that the input and output are not dependent. This creates a parallel architecture where client can take inputs without depending on the client handler’s output.

Multi-Threading has also been implemented in the client handler. As we have made the input in the client independent, we need to make it capable of handling multiple instructions simultaneously and one instruction execution should not be dependent on the other. In order to achieve this, I have created a separate thread for each command making it able to execute multiple commands in parallel hence removing the dependency and making it more memory efficient. As we cannot predict the scheduling of the threads, it is normal to expect results not in order.

**Signal Handling:**

Another important technique is the signal handling which is necessary to keep track of all your child processes and to avoid abnormal termination. In the case of keeping a list of every child process executed by the user or having a list of active clients, signal handling is required to update the list entries upon termination of the processes. Abnormal Termination also needs to be handled such as SIGINT signal received by the server when clients are still connected. It is our responsibility to handle these signals and act accordingly. I have implemented multiple signal handlers in these three programs based on the requirements and to avoid abnormal termination.

**Concurrency Control/Mutex:**

Another problem that is faced when using multi-threading is to avoid the race condition when a global variable or a shared resource is accessed from a thread. As thread scheduling is not predictable, it is necessary to protect that resource from uncertain modifications. To ensure protection from such behavior we use Mutexes. As in our case, the list needs to be global so that it is gets updated if a child process terminates or a new one executes. I have implemented a mutex lock when updating the list so that my critical code gets executed without any interference or change in resource ownership. It is important to unlock the resource once the values are updated. Signal handlers are an exception as using mutex in a signal handler can cause a deadlock!