Main basic idea: The server will execute commands sent by clients and return the execution results. To handle multiple clients concurrently, it utilizes a message queue for initial connection requests and named pipes for the actual command transmission and result reception. For each connected client, the server spawns a separate child process, allowing multiple client requests to be processed in parallel.

The phrase "for initial connection requests" refers to the use of a message queue as the initial communication channel between clients and the server. Clients send a request to connect to the server through this message queue. Once the server accepts the connection request, further communication regarding the sending of commands and receiving of results is handled through named pipes.

For each client-server child pair, clients are responsible for creating these pipes using the **mkfifo()** system call and provide their names upon connection to the server. The client can receive command lines either interactively from a user or read them from a file, which are then sent to the server for execution via the cs pipe.

In the context of your project, "processes" refer to instances of running programs, which in this case are the server and client applications. A message queue is a form of inter-process communication (IPC) that facilitates message exchange between these processes. Specifically, it allows the server process to receive incoming connection requests from multiple client processes.

These functions are part of the POSIX API (Portable Operating System Interface), which is implemented in Unix-like operating systems including Linux.

Why child server creates running child processes even if the commands sent from a single client are to be executed sequentially?

1. **Simplicity in Handling Output**: Redirecting output to a temporary file or handling pipes between commands becomes simpler when each command runs in its own process. The redirection of input/output streams (using **dup2()** for example) is done once per command execution without affecting the server's child process's standard input, output, and error streams.
2. **Concurrency and Synchronization**: Although each server's child process handles commands sequentially from a single client, the server as a whole can manage multiple clients concurrently. Using runner child processes allows each server's child process to wait for a command's execution to complete (using **wait()** or **waitpid()**) before proceeding to the next command, ensuring that commands are executed in the correct order and that output is correctly captured and returned to the client.
3. **Use of exec() Family Functions**: The **exec()** family of functions replaces the current process image with a new process image. If the server's child process were to directly execute a command using **exec()**, it would cease to exist in its original form and couldn't perform further tasks like sending back the execution result to the client. Using a separate runner child process for each command execution prevents this issue.

Upon startup, the server initiates a message queue with **mq\_open()** using the **O\_CREAT** flag, then enters a loop waiting for client connection requests with **mq\_receive()**. Upon receiving a request, it extracts the client ID, names of the client-created named pipes, and the **WSIZE** value from the message. A child process is then spawned using **fork().**

To set up and use your **comserver** and **comcli** programs, you would typically follow these steps:

1. **Starting the Server (comserver)**: First, start the **comserver** program by specifying the message queue name (**MQNAME**) as an argument. This initializes the server, creating the message queue for incoming connection requests. The server then waits for clients to send their connection requests through this queue.

bashCopy code

./comserver MQNAME

1. **Running the Client (comcli)**: Next, run one or more instances of the **comcli** client program, using the same **MQNAME** to ensure it connects to the correct server. You can run the client in its default interactive mode or in batch mode by specifying a command file with the **-b COMFILE** option. If you wish to control the write size to the server, you can also include the **-s WSIZE** option.
   * **Interactive Mode**: The client waits for user input for command lines to execute. Each command entered by the user is sent to the server for execution, and the output is returned to the client and displayed.

bashCopy code

./comcli MQNAME

* + **Batch Mode**: If you have a set of commands in a file that you wish to execute sequentially, you can start the client in batch mode by specifying the file name with the **-b** option.

bashCopy code

./comcli MQNAME -b COMFILE

1. Optionally, specify the **-s WSIZE** to control the write size to the server's sc pipe.
2. **Expected Behavior and Output**:
   * The server, upon receiving a connection request, creates a child process to handle communications with the connected client.
   * When the client sends a command, the server's child process executes the command and sends the output back to the client through the named pipes.
   * The client displays the received output to the user.
   * This process repeats for each command entered in interactive mode or read from a file in batch mode.
   * Clients can disconnect by sending a quit command, at which point the server's child process for that client will terminate.

To implement these message types in your project, follow these steps:

1. **Define a Struct for Message Format**: Create a C struct that includes the fields for length, type, padding, and data. This struct will serve as the template for all your messages.
2. **Serialization and Deserialization**: Write functions to convert your message struct to a byte stream (serialization) before sending, and to parse a byte stream back into a struct (deserialization) upon receiving. This is crucial for transmitting messages through pipes and message queues.
3. **Sending and Receiving Messages**:
   * For message queues, use **mq\_send()** and **mq\_receive()** with the serialized message data.
   * For named pipes, use **write()** and **read()** on the file descriptors associated with your pipes, handling the data in chunks if necessary, especially for the **COMRESULT** messages that may exceed **WSIZE**.

To specify the message type in your messages, you define a part of your message structure to hold the message type identifier. This is often a single byte (or an integer) that represents different types of messages (e.g., **CONREQUEST**, **COMLINE**, etc.). When constructing a message, you set this part of the structure to the appropriate value indicating the message type before sending the message.

For example, in C:

cCopy code

struct message { uint32\_t length; // 4 bytes for length uint8\_t type; // 1 byte for message type uint8\_t padding[3]; // 3 bytes of padding for alignment char data[]; // Variable length data };

When sending a message, you populate the **type** field with the constant that represents your message type, like so:

cCopy code

struct message msg; msg.type = CONREQUEST; // Set the message type

Ensure each message type (e.g., **CONREQUEST**, **CONREPLY**, **COMLINE**) is predefined as constants, and the receiving end knows to check this field to determine how to process the message.