# CLIENT-SERVER ARCHITECTURE

**Gaurav Singh**
**MCA 2018-21**

## ABOUT THE PROJECT :

The goal of this poject is to implement a TCP client server application to implement a service that returns the sounds made by animals. The name of each requested animal will be specified by a client whose process you will also create. The server will start in *passive* mode listening on a specified port for a transmission from a client. Separately, the client will be started and will contact the server on a given IP address and port number that must be entered via the command line.

**For communication with server client has to register itself with server, firstly client have to log in with id and password. If id and password of client is correct then client can communicate with server otherwise server sends a "Sorry your id and password is not correct" response to the client.**

**The server, which will be called SoundServer, will operate as follows:**

- Return the sound made by animals named by a client after the client connects to the server using the SOUND instruction,
- Accepts a STORE message used for storing new (animal, sound) pairs,
- Accepts a QUERY message used to ask which animals it knows,
- Accepts a BYE message that results in closing of the current session between the client and the server.
- Accepts an END message that results in closing of the current session and termination of the server.

**Operation of the client and the server is as follows:**

When the SoundServer server starts, it initially knows about five pairs each containing an animal's name and the sound it makes, e.g. 'dog' and 'woof'. The server stores this start-up data in an internal data structure of size sufficient to store the initial five pairs, plus up to another ten pairs. The server then waits for a client to connect to it using a server port number specified by you when you start the server. The server is required to concurrently interact with clients. In other words, if the server is interacting with a client, it can interact with other clients also at the same time.

The client (called SoundClient) uses SOUND, STORE, QUERY, BYE and END messages to instruct the server. The client program will be capable of sending any of these messages on instruction from the user.

**Operation of the client-server pair is as follows:**

Note that, regardless of the case of characters typed by the user of the SoundClient, the characters are sent to the SoundServer as upper case.

**SOUND:** This message is sent from the client to the server to make the client's connection to the server, thus initiating a session. It contains the ASCII string “SOUND” followed by the newline character “\n”. On sending the SOUND message, the client waits for a return message from the server via the socket that connects them. After receiving and displaying the return message the client loops back so that the user can initiate sending of another message.

On receipt of a SOUND message from a client, the server returns the string “SOUND: OK” followed by a newline character and waits for further instructions from the client. The message sequence to start a calculation session is as follows:

**Client: SOUND**

**Server: SOUND: OK**

Requesting sounds is performed when the client sends a string, representing the name of an animal, terminated by a newline character. On receipt of such a string, the server returns the sound made by the animal specified by the client, or indicates that it does not know that animal, as shown in the following examples.

**Client: DOG**

**Server: A DOG SAYS WOOF**

**Client: CAT**

**Server: I DON’T KNOW CAT**

**STORE:** This message is sent from the client to the server and causes the following animal name and sound made by that animal (i.e. two arguments to the STORE instruction) to be stored in the server’s memory. The exchange is started by the sending of the ACSII string “STORE” followed by the newline character. Following this is the name of the animal to be stored, a newline character, and then the sound made by that animal. The server responds with the “STORE: OK” message to confirm that the store operation has been completed.

A typical message sequence would be as follows:

**Client: STORE**

**CAT**

**MEOW**

**SERVER: STORE: OK**

If the server already knows the animal specified by the first argument, the sound specified by the second argument replaces that stored by the server. If the server’s storage is full (i.e. it already knows 15 animals and the sound they make), and the newly-specified animal is not

already known to the server, the server returns STORE: OK but does not actually store the newly defined data.

**QUERY:** This message is sent from the client to the server, and causes the server to return the names of all the animals stored in the server's memory. The exchange is started by the client's sending of the ACSII string "QUERY" followed by the newline character. The server responds with a sequence of newline-terminated animal names, followed by the "QUERY: OK" message to confirm that the query operation has been completed.

**A typical message sequence would be as follows:**

**Client: QUERY**

- **SERVER: DOG**
- **HORSE**
- **SNAKE**
- **COW**
- **SHEEP**
- **CAT**
- **QUERY: OK**

**BYE:** This message is sent from the client to the server, instructing the server that the client no longer needs the current session (that was initially created by the SOUND message). The client sends the ASCII string "BYE" followed by a newline character. After reading and displaying the returned message from the server, the client should close its connection. The server, on receipt of the BYE message, should return the string "BYE: OK" followed by a newline character, and then should wait for a connection from a new client. Note that any animal-sound pairs stored by the old client are retained in the server's memory and can be requested by any new client.

A typical message sequence would be as follows:

**Client: BYE**

**SERVER: BYE: OK**

**END:** This message is sent from the client to the server, instructing the server to shutdown. The client sends the ASCII string "END" followed by a newline character. After reading and displaying the returned message from the server, the client should close its connection. The server, on receipt of the END message, should return the string "END: OK" followed by a newline character, then should close all open sockets and terminate. Note that any animal sound pairs added since the server started are not saved, and if the server is subsequently restarted, it reverts to knowing only about the initial five pairs.

A typical message sequence would be as follows:

**Client: END**

**Server: END: OK**