Client-Server architecture and corresponding UML class diagram

Client-server architecture

- Distributed system model which shows how data and processing is distributed across a range of components, but can also be implemented on a single computer.
- Set of stand-alone servers which provide specific services such as printing, data management, etc.
- Set of clients which call on these services.
- Network which allows clients to access servers.

A system that follows the client–server pattern is organized as a set of services and associated servers, and clients that access and use the services. The major com- ponents of this model are:

1. A set of servers that offer services to other components. Examples of servers include print servers that offer printing services, file servers that offer file man- agement services, and a compile server, which offers programming language compilation services.
2. A set of clients that call on the services offered by servers. There will normally be several instances of a client program executing concurrently on different computers.
3. A network that allows the clients to access these services. Most client–server systems are implemented as distributed systems, connected using Internet protocols.

Client–server architectures are usually thought of as distributed systems architectures but the logical model of independent services running on separate servers can be implemented on a single computer. Again, an important benefit is separation and independence. Services and servers can be changed without affecting other parts of the system.

Clients may have to know the names of the available servers and the services that they provide. However, servers do not need to know the identity of clients or how many clients are accessing their services. Clients access the services provided by a server through remote procedure calls using a request-reply protocol such as the http protocol used in the WWW. Essentially, a client makes a request to a server and waits until it receives a reply.

Overview of Client-Server architecture pattern:

Description: In a client-server architecture, the functionality of the system is organized into services, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them.

When used: Used when data in a shared database has to be accessed from a range of locations. Because servers can be replicated, may also be used when the load on a system is variable.

Advantages: The principal advantage of this model is that servers can be distributed across a network. General functionality (e.g., a printing service) can be available to all clients and does not need to be implemented by all services.

Disadvantages: Each service is a single point of failure so susceptible to denial of service attacks or server failure. Performance may be unpredictable because it depends on the network as well as the system. May be management problems if servers are owned by different organizations.

To further clarify “Client-Server” architecture, let's explore a specific example.
Client-Server example:
Application Description: The **Library Management System** is a desktop application for librarians, enabling them to manage book collections, user information, and borrowing transactions. Operating with a thick client architecture, the application handles all business logic locally, while a server database stores and retrieves data.

Functional Requirements

1. **Book Management**: Add, update, and delete book records, including details such as title, author, ISBN, and availability status.
2. **User Management**: Register new users, update user information, and manage user records.
3. **Transaction Management**: Facilitate book borrowing and returning, tracking user and book details for each transaction.
4. **Data Storage**: Store and retrieve all book, user, and transaction data in a centralized server database.

Class diagram in plantUML of the application with client-server architecture pattern:

```
@startuml
package "Client" {
   class LibraryApp {
    - bookManager: BookManager
    - userManager: UserManager
    - transactionManager: TransactionManager
    + run()
   }

class BookManager {
    + addBook(title: String, author: String, isbn: String)
    + updateBook(bookId: Int)
    + deleteBook(bookId: Int)
   }

   class UserManager {
    + addUser(name: String, contactInfo: String)
    + updateUser(userId: Int)
    + deleteUser(userId: Int)
   }

   class TransactionManager {
    + borrowBook(userId: Int, bookId: Int)
    + returnBook(transactionId: Int)
   }

   LibraryApp --> BookManager
   LibraryApp --> UserManager
   LibraryApp --> TransactionManager
}

package "Server" {
   class Book {
    - bookId: Int
    - title: String
```

```
    - author: String
    - isbn: String
    - availabilityStatus: Boolean
  }

  class User {
    - userId: Int
    - name: String
    - contactInfo: String
  }

  class Transaction {
    - transactionId: Int
    - bookId: Int
    - userId: Int
    - transactionDate: Date
  }

  interface Database {
    + saveData(data: String)
    + fetchData(query: String): String
  }

  Database <|-- Book
  Database <|-- User
  Database <|-- Transaction
}

BookManager --> Server.Database
UserManager --> Server.Database
TransactionManager --> Server.Database

@enduml
```