

# Executive Summary

## Assignment Overview:

Implementation of the Remote Procedure Calls and Multithreading for a storage application. The client and server will try to connect over RPC and communicate the data. For RPC communication, I have used RMI (Remote Method Invocation) which is a mechanism that allows one Java Virtual Machine (JVM) to invoke methods on an object running in another JVM. It functions by allowing remote communication between programs written in Java.

Multithreading helps to run multiple threads of a process and increases the concurrency in the application also brings the issue of handling the synchronization in the system. In the application, I handled the issue of synchronization by setting the critical section for multiple data key stores. And threads can access different resources parallelly if there is no conflict with the critical section. The resources are being utilized with the help of RPC communication between client and server. The RMI library is used to register the shared resource and lets the client access the data resource provided by the client.

## Technical Overview:

Remote Procedure Calls (RPC) are a protocol that allows a computer program to cause procedures to execute on another address space i.e., on another physical machine. This protocol abstracts complexity, enabling a seamless communication process between client and server. In RPC, a client sends a request message to a remote server to execute a specified procedure using the arguments supplied. The server responds by executing the function and sending the results back to the client.

Multithreading in client-server communication allows simultaneous processing of multiple client requests, enhancing performance and responsiveness. By allocating separate threads for each request, servers can handle interactions concurrently, preventing bottlenecks associated with single-threaded models. This concurrency optimizes resource utilization, with threads sharing memory space within the same process, ensuring efficient communication and quicker response times.

Moreover, multithreading ensures fault tolerance, if one thread fails, others continue, preventing server-wide disruptions. However, despite these advantages, multithreading introduces complexity, necessitating careful programming to avoid synchronization issues, deadlocks, and resource contention. I have handled this using a log which keeps track of the key and value if a thread uses it.

request processing. This enhances concurrency, leading to faster response times and optimal resource utilization. Multi-threading allows clients to interact independently, even when one client's operation is ongoing.

## READ ME

### Assumptions:

- Key entered for the store should be string and should not contain any spaces in between.
- Value entered for the store should be integer.
- Time to live for the client expecting response from the server is 5 seconds.
- Client sends a unique message id which is appended in the request message. To handle unrequested packet.
- All user input commands should be in uppercase (PUT/DELETE/GET). Server will say Invalid command if user tries to enter otherwise.
- Store is initialized with seed data of multiple values.

### How to start server application:

- start the serverApplication file which provides the RMI to the client and registers in the registry.

### How to start client application:

- UI can be started with different modes "ui", "thread", "exit"
- ui takes to interactive mode.
- thread runs multiple thread of client to run the application.
- exit quits the client application.

### Steps to use:

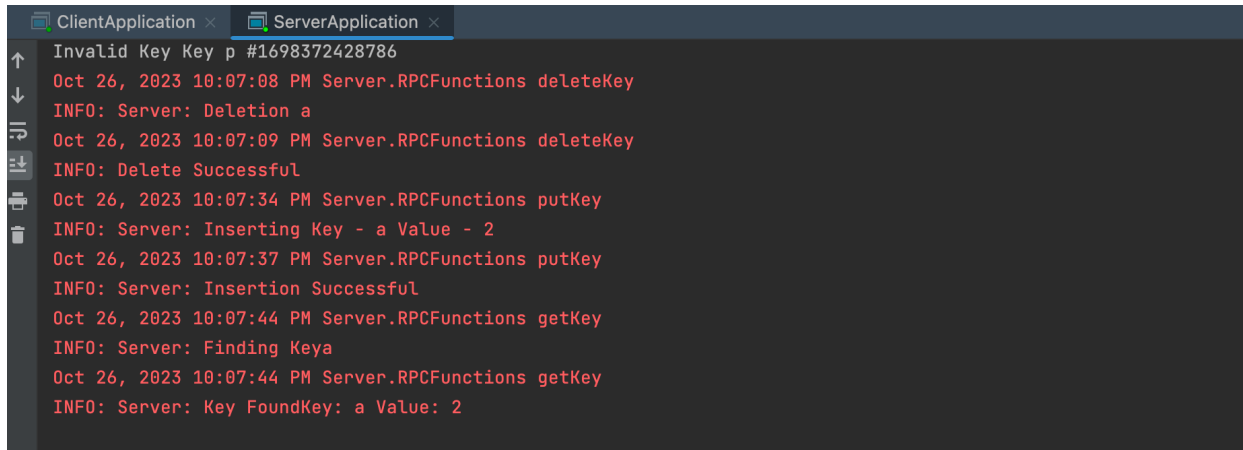
- Every time an operation is performed on the client or server, the log will be ClientLog.log and ServerLog.log in corresponding packages.
- Please enter the user commands in the client application terminal.

```
PUT a 2  
GET a  
DELETE a
```

Implementing this protocol helped to comprehend network communication, highlighting the trade-offs between reliability and speed. Integrated the logger for logging communication activities, which helped to maintain a comprehensive record of communication between the

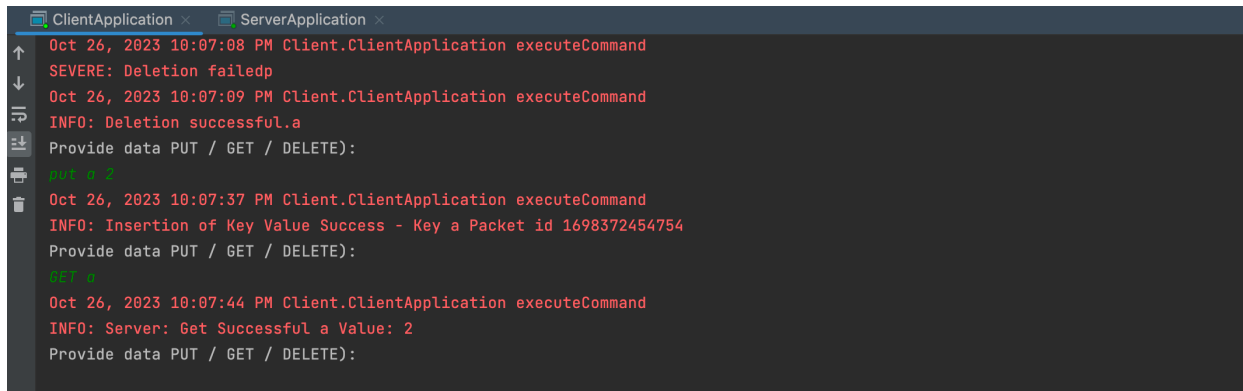
client and server. Logging not only aided in debugging but also provided insights into the sequence of events during execution.

## SERVER APPLICATION



```
ClientApplication x ServerApplication x
Invalid Key Key p #1698372428786
Oct 26, 2023 10:07:08 PM Server.RPCFunctions deleteKey
INFO: Server: Deletion a
Oct 26, 2023 10:07:09 PM Server.RPCFunctions deleteKey
INFO: Delete Successful
Oct 26, 2023 10:07:34 PM Server.RPCFunctions putKey
INFO: Server: Inserting Key - a Value - 2
Oct 26, 2023 10:07:37 PM Server.RPCFunctions putKey
INFO: Server: Insertion Successful
Oct 26, 2023 10:07:44 PM Server.RPCFunctions getKey
INFO: Server: Finding Keya
Oct 26, 2023 10:07:44 PM Server.RPCFunctions getKey
INFO: Server: Key FoundKey: a Value: 2
```

## CLIENT APPLICATION



```
ClientApplication x ServerApplication x
Oct 26, 2023 10:07:08 PM Client.ClientApplication executeCommand
SEVERE: Deletion failedp
Oct 26, 2023 10:07:09 PM Client.ClientApplication executeCommand
INFO: Deletion successful.a
Provide data PUT / GET / DELETE):
put a 2
Oct 26, 2023 10:07:37 PM Client.ClientApplication executeCommand
INFO: Insertion of Key Value Success - Key a Packet id 1698372454754
Provide data PUT / GET / DELETE):
get a
Oct 26, 2023 10:07:44 PM Client.ClientApplication executeCommand
INFO: Server: Get Successful a Value: 2
Provide data PUT / GET / DELETE):
```

