Assignment 2

CS G623 - Advanced Operating Systems

DataNexus

https://github.com/Rohan-Witty/DataNexus

-By
Shashank Shreedhar Bhatt - 2020A7PS0078P
Rohan Srinivasan - 2020A7PS0081P

Introduction

Welcome to the technical documentation for "DataNexus" a robust database system with support for concurrent, distributed access in the form of reads, edits, and writes while maintaining perfect consistency.

DataNexus illustrates different communication paradigms: TCP, UDP, and Java Remote Method Invocation. This documentation serves as your gateway to understanding the intricacies of the protocols followed by these protocols, enabling you to dive into the heart of the database system's workings and architecture.

About DataNexus

"DataNexus" is a dynamic multiuser database system that allows users to access a centralised database.

As a user or client, you can perform multiple operations on the database: `put`, `get`, `delete`, `clean`, `store`, and `exit`.

Dependencies

Before delving into the code, it's important to note the dependencies required to run this script successfully. The code relies on the following libraries and technologies:

- Java 17+
- Java RMI package

Please ensure that you have these dependencies installed before running the script.

Design Choices

Database in the form of a text file

Our centralised database is in the simplest possible format - a simple, space-separated text file with a new record in each row. For deletion, we use a strategy similar to a dirty bit, where the record is not immediately removed from the file. Just the first character is set to a blank space.

Mutual Exclusion through exclusive *Java Reentrant Locks*

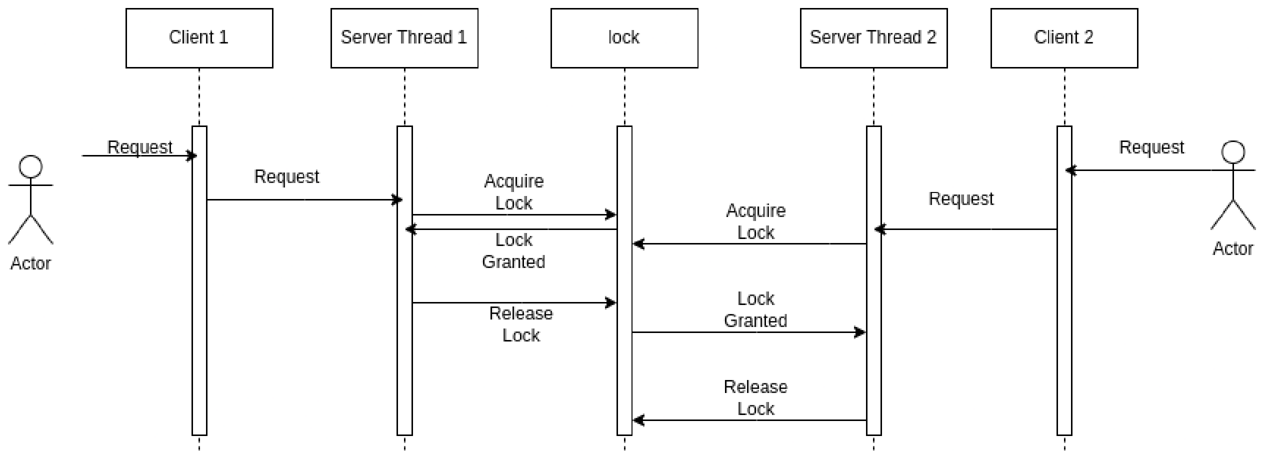


Figure 1: Working of Reentrant Locks in our programs

The design choice to use a single exclusive lock and no separate read-write

Here, we realised that if we want to maintain the order of operations and exclude the possibility of stale reads, we decided to go forward and keep a single lock for any operation on the database. Calls to `processRequest()` in all three implementations are made only after acquiring the lock, which ensures that mutual exclusion is maintained.

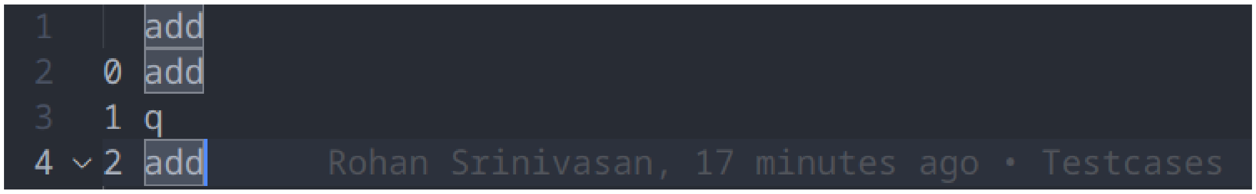
The Java Reentrant lock, when given the fairness parameter as `true`, behaves fairly and awards the lock to the longest waiting thread. This ensures that FIFO ordering is maintained. This is a centralised mutual exclusion algorithm.

Methodology followed for updates and deletes

Design choice to give speed of operations more priority over size of the file

In Java, there is no way to replace a line in a file with another line in a reliable manner, nor is there any way to delete a line, both without having to rewrite an entire file. We decided to forgo rewriting the entire database each time there is an update or delete request, as in the case of a very large database, the time taken to process the request would be high. Instead, we are appending the updated value at the end of the file (in the case of updates) and adding a designated signature to outdated lines. Although this adds extra lines to the database each time there is a delete or update request, this reduces the time taken for each access. To resolve the issue of expanding the text database, we have also added a new feature.

Design choice to add blank character ' ' at the start of updated/deleted line



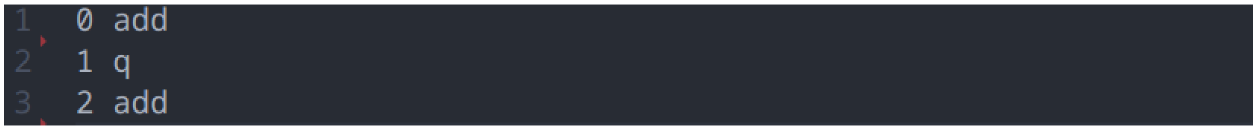
```
  add
0 add
1 q
2 add      Rohan Srinivasan, 17 minutes ago • Testcases
```

Figure 2: Example showing the *database.txt* file where the first entry was deleted

Figure 2 is an example of a key that has been updated. The same convention is followed for deleted entries. The value associated with key '1' has been modified from 'add' to 'q'. It would look like this when the key on line 1 was 4 (say) instead and the entry was deleted. Later, a call to a functionality that we added can be made to clean it up.

The designated signature, indicating that the line has been modified, is a blank (' ') character. On reading a blank character at the start of a line, the line is treated as an outdated value and is skipped.

Additional functionality to clean the database



```
0 add
1 q
2 add
```

Figure 3: Example showing the *database.txt* file after clean was called on Figure 2

Figure 3 is an example of what happens when the added functionality is called. Refer to the example of an updated/deleted line provided in Figure 2 from the previous section.

Since we chose to give speed priority over space, we also wanted to provide a way of overcoming the disadvantage that the tradeoff provides. We added a new `clean` operation that can be carried out on the database. It removes all the outdated values in the text database and cleans it up. This way, the clients accessing the database can mix in the clean commands to clean up the database based on their own needs. This is also automatically called when the `store` feature is used.

Connection Closing Mechanism

The client sends an `exit` to the server. The server does not set its `connected` variable to false and sends a `Goodbye` message to the client. The client, on seeing this, can shut itself down.

In TCP, we are trying to close as many connections as possible. Here, on receiving the `exit` command, it waits for 10 seconds to close any other client connections that come in with a request. After that, the server closes itself.

Command Line Arguments

We have made use of command line arguments to distinguish between server and client, as well as to determine additional behaviour semantics for the client.

Client vs Server

To initialise the server while running the Java script, an additional command line argument `server` must be passed. If a `server` command line argument is not passed, then the client code is run.

IP Address of Server

In case we need to specify the IP address of the server when the server is not running on the same machine,

Performance comparison

For comparing the speeds, we programmatically generated a test case file containing 1000 requests and made them from a client to a server in all three types. Each one was run starting with an empty database for fairness. We have included the Python code for generating this file in our repository in the file `generate_dataset.py`. Now, each of the files was modified to optionally run in the test case mode, where the latencies are measured and logged for running each command. These were analysed using a spreadsheet, and the following average time statistics were observed (in ns):

Command	Time in TCP	Time in RMI	Time in UDP
del	105442	1803602	2724446
get	88754	1734589	2552406
put	97539	1841216	2793350
store	122689	1813908	2737827
exit	7194	1203722	735643
Total	103161	1799941	2705032

Table 1: Command wise performance in nanoseconds in the single client setting

Note that here in the test case file, the `exit` command is present only once at the end, while others are present randomly and thus uniformly distributed (approximately 250 occurrences each). Here, the same data is visualised in a graph where the X-axis is the command and the Y-axis is the average time taken in nanoseconds. It is observed that, on average, TCP is **17** times faster than RMI and **27** times faster than UDP. This is probably because of the time saved because of connection reuse.

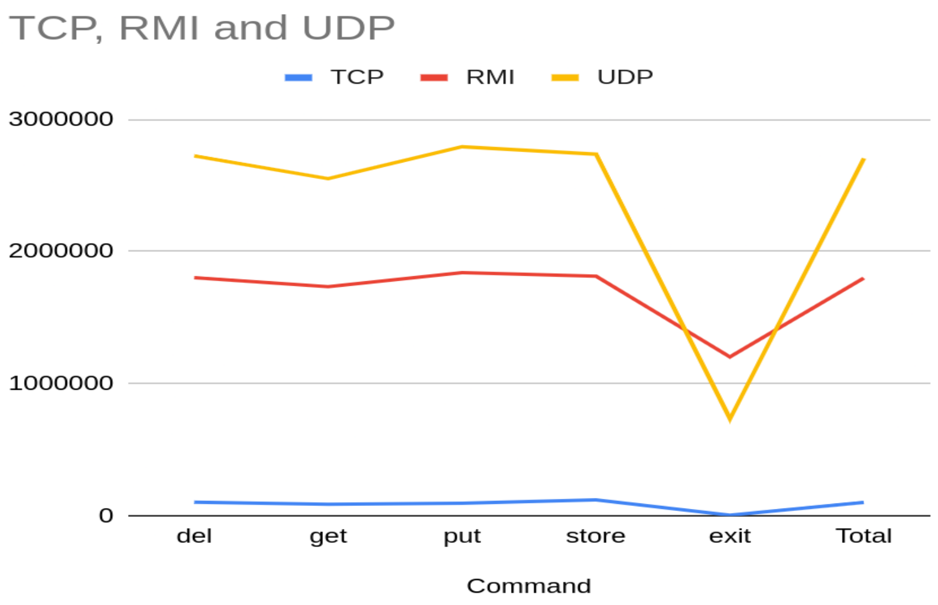
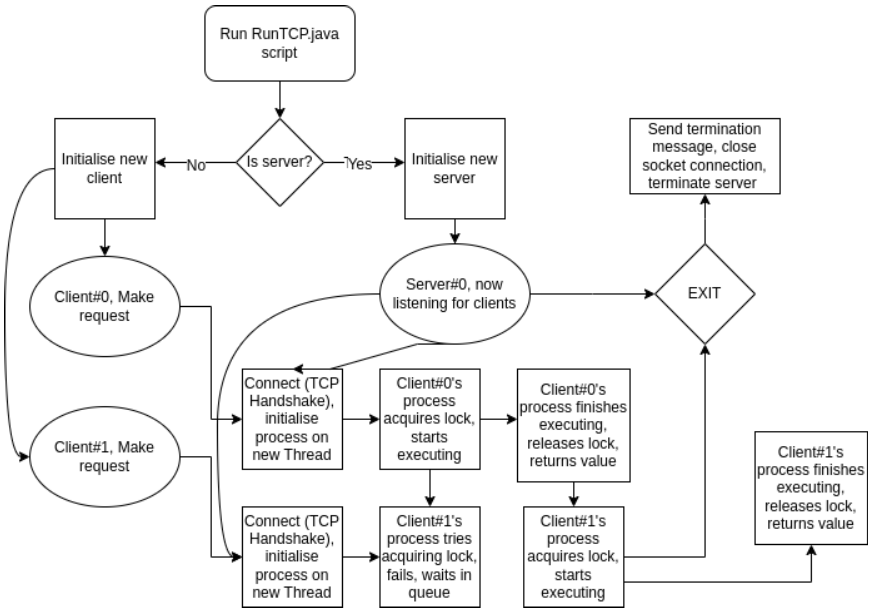


Figure 4: Illustration of Table 1, comparing running times of TCP, RMI and UDP in the single client setting

Execution flow

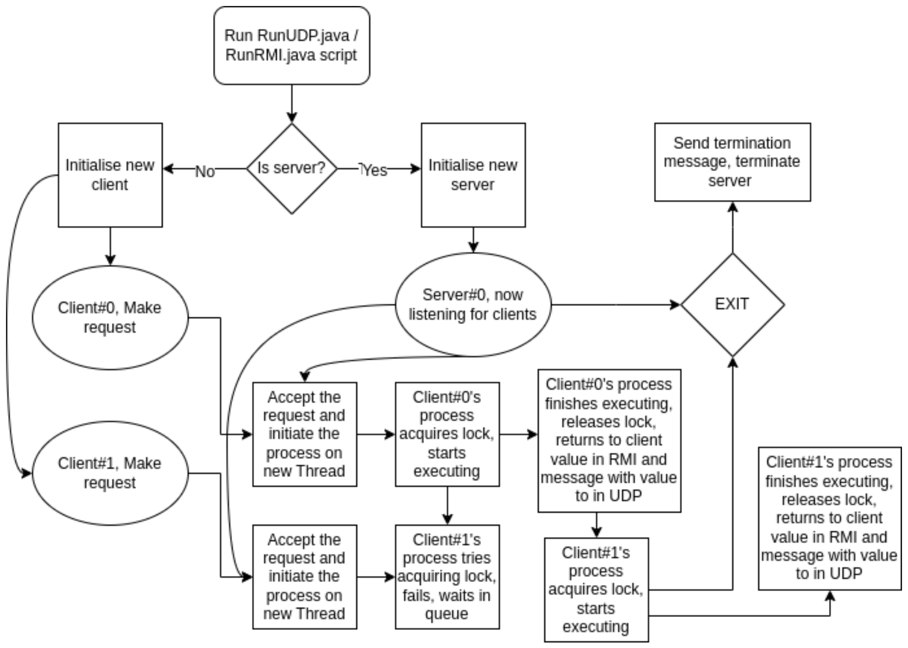
TCP



Run RunTCP.java script
Is server?
No
Yes
Initialise new client
Initialise new server
Client#0, Make request
Client#1, Make request
Server#0, now listening for clients
EXIT
Send termination message, close socket connection, terminate server
Connect (TCP Handshake), initialise process on new Thread
Client#0's process acquires lock, starts executing
Client#0's process finishes executing, releases lock, returns value
Connect (TCP Handshake), initialise process on new Thread
Client#1's process tries acquiring lock, fails, waits in queue
Client#1's process acquires lock, starts executing
Client#1's process finishes executing, releases lock, returns value

Figure 5: A flowchart representing the execution flow of the TCP programme

UDP/RMI



Run RunUDP.java / RunRMI.java script
Is server?
No
Yes
Initialise new client
Initialise new server
Client#0, Make request
Client#1, Make request
Server#0, now listening for clients
EXIT
Send termination message, terminate server
Accept the request and initiate the process on new Thread
Client#0's process acquires lock, starts executing
Client#0's process finishes executing, releases lock, returns to client value in RMI and message with value to in UDP
Accept the request and initiate the process on new Thread
Client#1's process tries acquiring lock, fails, waits in queue
Client#1's process acquires lock, starts executing
Client#1's process finishes executing, releases lock, returns to client value in RMI and message with value to in UDP

Figure 5: A flowchart representing the execution flow of the UDP and RMI programmes

Components

RunTCP.java

Overview

This file implements a basic TCP communication system in Java, capable of running in two modes: as a TCP server or as a TCP client. The system allows for sending and receiving messages, managing a simple key-value store, and handling multiple client connections concurrently.

Functionality

- **Server Mode:** Listens for client connections and processes incoming requests such as storing, retrieving, and deleting key-value pairs in a database and managing server operations.
- **Client Mode:** Connects to the server, sends messages or commands, and receives responses. It includes a testing mode for measuring TCP latency.

Code Structure

- **RunTCP:** The entry point of the application, responsible for initiating the server or client based on command-line arguments.
- **TCPClient:** Handles the client-side operations, including connecting to the server, sending messages, and receiving responses.
- **TCPServer:** Manages the server-side operations, listens for client connections, and spawns threads (ClientHandler) to handle them.
- **ClientHandler:** A thread class responsible for handling individual client connections to the server.

Key Components

RunTCP Class:

- **main() Method:** Interprets command-line arguments to start either the server or client.
- **Decision Logic:** Determines the mode of operation (server or client) based on the presence of the "server" argument.

TCPClient Class:

- **Connection Establishment:** Connects to the server using a specified IP address.
- **Message Sending and Receiving:** Handles the input and output streams for communication with the server.
- **Testing Mode:** Includes a mode for testing TCP latency by sending predefined messages and recording response times.

TCPServer Class:

- **Server Socket Initialization:** Creates a server socket and listens for client connections.
- **Client Handling:** Accepts client connections and starts a new ClientHandler thread for each connection.
- **Concurrency Management:** Utilises a ReentrantLock for managing concurrent access to shared resources.

ClientHandler Class:

- **Client Communication:** Manages the input and output streams for each client connection.
- **Request Processing:** Processes various commands like "put", "get", "del", "store", and "clean" for managing a simple key-value store.
- **Concurrency Control:** Employs a lock to ensure thread-safe operations on shared resources.

Database Management:

- **File-Based Storage:** Utilises a text file ("database.txt") for storing key-value pairs.
- **CRUD Operations:** Implements create, read, update, and delete operations on the key-value store.

Error Handling:

- **Exception Management:** Catches and handles IOExceptions to ensure robustness.

RunUDP.java

Overview

This file implements a basic UDP communication system in Java, capable of operating in two modes: as a UDP server or as a UDP client. The system facilitates sending and receiving messages, managing a simple key-value store, and handling multiple client requests.

Functionality:

- **Server Mode:** Listens for client requests and processes incoming commands such as storing, retrieving, and deleting key-value pairs in a database.
- **Client Mode:** Sends messages or commands to the server and receives responses. Includes a testing mode for measuring UDP latency.

Code Structure

- **RunUDP:** The main class that initiates the server or client based on command-line arguments.
- **UDPClient:** Manages the client-side operations, including sending messages and receiving responses.
- **UDPServer:** Handles server-side operations, listens for client requests, and processes them.
- **DatagramHandler:** A thread class responsible for handling individual client requests in the server.
- **Receiver:** A thread in the client that listens for responses from the server.

Key Components

RunUDP Class:

- **main() Method:** Determines whether to start the server or client based on command-line arguments.

UDPClient Class:

- **Connection and Communication:** Handles the creation of a DatagramSocket and sending/receiving of DatagramPackets.
- **Testing Mode:** Includes functionality for sending test messages and recording response times for latency measurement.

UDPServer Class:

- **Server Initialization:** Sets up a DatagramSocket to listen for client requests.
- **Request Handling**: Receives DatagramPackets and starts a new DatagramHandler thread for each request.

DatagramHandler Class:

- **Request Processing**: Processes commands like "put", "get", "del", "store", and "clean" for a simple key-value store.
- **Response Sending**: Sends back responses to the client after processing each request.

Receiver Class (Client-Side):

- **Response Listening:** Continuously listens for responses from the server and prints them to the console.

Database Management:

- **File-Based Storage:** Utilises "database.txt" for storing key-value pairs.
- **CRUD Operations:** Implements create, read, update, and delete operations on the key-value store.

Error Handling:

- **Exception Management:** Catches and handles IOExceptions and InterruptedExceptions.

RunRMI.java

Overview

This file implements a basic Remote Method Invocation (RMI) communication system in Java. It allows for remote procedure calls between a client and a server, facilitating operations like storing, retrieving, and deleting key-value pairs.

Functionality:

- **Server Mode:** Hosts an RMI service that listens for client requests and processes commands such as "put", "get", "del", "store", and "clean".
- **Client Mode:** Connects to the RMI server and sends requests, receiving responses from the server.

Code Structure

- **RunRMI:** The main class that initiates the server or client based on command-line arguments.
- **RemoteCommand:** Implements the Command interface and handles the server-side processing of client requests.
- **Command:** An interface defining the remote method processRequest.
- **Terminate:** A thread class is used to terminate the server after a delay.

Key Components

RunRMI Class:

- **main() Method**: Determines whether to start the server or client based on command-line arguments.
- **startServer Method**: Sets up the RMI registry and binds the Command stub.
- **startClient Method**: Looks up the Command stub from the registry and processes user commands.

RemoteCommand Class:

- **RMI Server Logic:** Implements methods for processing client requests like "put", "get", "del", etc.
- **Thread Safety:** Uses a ReentrantLock to ensure thread safety in concurrent environments.

Command Interface:

- **Remote Method Declaration:** Defines the processRequest method for remote invocation.

Terminate Class:

- **Server Shutdown:** Provides a mechanism to gracefully shut down the server.

Database Management: Manages a simple file-based key-value store.

- **File-Based Storage:** Utilises "database.txt" for storing key-value pairs.
- **CRUD Operations:** Implements create, read, update, and delete operations on the key-value store.

Error Handling:

- **Exception Management:** Catches and handles RemoteExceptions and IOExceptions.