Advanced IT - Testat 3

Aufgabenstellung

In dieser Aufgabe soll ein File-Server für Textdateien entwickelt werden. Vereinfachend gehen wir davon aus, dass dem Server ein festes, bereits existierendes Basisverzeichnis zugeordnet ist, in dem sich alle verwalteten Dateien befinden und dass er die notwendigen Zugriffsrechte besitzt. Die Textdateien sind dabei zeilenweise organisiert und beginnen mit Zeilennummer 1. Der Server soll als **Worker-Pool-Server** auf **Port 5999** Aufträge in Form von Strings mit `READ filename,line no` entgegennehmen, wobei line no eine positive ganze Zahl sein muss. Daraufhin wird vom Server die Datei filename geöffnet, die Zeile line no ausgelesen und zurückgeschickt. Außerdem soll der Server auch das Kommando `WRITE filename,line no,data` verstehen, bei dem die Zeile line no durch data (kann Kommas und Leerzeichen enthalten) ersetzt werden soll. Falls sich im Basisverzeichnis des Servers keine solche Datei befindet oder keine entsprechende Zeile vorhanden ist, soll an den Client eine Fehlermeldung zurückgesendet werden.

Achten Sie darauf, dass nebenläufige Zugriffe konsistente Dateien hinterlassen. Implementieren Sie hierzu das Zweite Leser-Schreiber-Problem (mit Schreiberpriorität) mit dem Java Monitorkonzept!

Implementieren Sie den Server sowie einen kleinen Test-Client. Verwenden Sie Java und UDP!

Testen Sie die Nebenläufigkeit und das Einhalten der Schreiberpriorität durch geeignete Szenarien und dokumentieren Sie die Testfälle!

Umsetzung

Um das Ganze etwas dynamischer zu gestalten, habe ich mir die Freiheit genommen, den Server so zu implementieren, dass er einen entsprechenden Ordner erstellt, wenn der Benötigte nicht auf dem Desktop vorhanden ist. Sollte diese Funktionalität nicht erwünscht sein, so lässt sich die **entsprechende Zeile (65 - 74)** aus dem Code der Klasse `Server.java` entfernen. Die dann geworfene Fehlermeldung, sollte der Ordner fehlen, wird durch die Exceptions abgefangen.

Um den Anforderungen des **Worker-Pools** gerecht zu werden, werden durch den Server (`Server.java`) 5 Worker gestartet, welche die eingehenden Aufträge bearbeiten sollen.

Server.java

```
for(int i = 0; i < workers.length; i++) {
    workers[i] = new Worker(i + 1, serverSocket, requestQueue, monitor,
PATH);
    workers[i].start();
    System.out.println("SUCCESS: Worker "+ (i + 1) +" was started!");
}
```

In einer Endlosschleife fügt dabei der Server (`Server.java`) die eingehenden Aufträge der Clients zu einer Job-Warteschlange hinzu, welche durch eine LinkedList (`Queue.java`) realisiert wurde.

Server.java

```
while (true) {
    try {
        dp = new DatagramPacket(new byte[MAXSIZE], MAXSIZE);
        serverSocket.receive(dp);
        requestQueue.add(dp);

    } catch (Exception e) {
        System.err.println("ERROR: " + e + "\nATTENTION: Shutting down
server!");
        System.exit(1);
    }
}
```

Ist ein Worker (`Worker.java`) bereit einen Auftrag zu bearbeiten, so holt er sich diesen aus der Job-Warteschlange, bearbeitet diesen und sendet die Antwort zurück an den Auftragsgeber, also den Client (`Client.java`).

Worker.java

```
while (Server.running) {
        DatagramPacket dp = q.remove();
        System.out.println("SUCCESS: Worker " + this.id + " received the
job from the queue!");

        try {
            InetAddress clientAddress = dp.getAddress();
            int clientPort = dp.getPort();
            String command = new String(dp.getData(), 0, dp.getLength());
            String s = process(command);
            DatagramPacket sendDp = new DatagramPacket(s.getBytes(),
s.getBytes().length, clientAddress, clientPort);
            socket.send(sendDp);

        } catch (Exception e) {
            System.err.println("ERROR: " + e);
        }
}
```

Mit Hilfe eines File-Monitors (`FileMonitor.java`) für jedes File (verbunden über eine Map), wird sichergestellt, dass kein Worker parallel zu einem anderen Worker eine Datei bearbeiten kann. Paralleles Lesen wird jedoch gewährleistet. Solange ein Worker etwas aus einer Datei ausliest, darf jedoch nicht in diese geschrieben werden. Hierbei wurde die Schreiber-Priorität implementiert.

FileMonitor.java

```
public synchronized void startRead() {

    while ((activeW) || (anzWaitingW > 0)) {
        try {
            this.wait();
```

```
        } catch (Exception e){
            System.err.println("ERROR: " + e);
        }
    }
    anzActiveR++;
}
```

Beispiele

Da das Prinzip des Beschreibens einer Datei sowie des Lesens aus ihr bereits aus der vorherigen Testataufgabe sowie durch die Übungen innerhalb der Vorlesung bereits gut etabliert wurde, wird innerhalb dieser Dokumentation auf das Zeigen der verschiedenen Fälle mittels Groß- und Kleinschreibung von Kommandos verzichtet. Mittels

```
if (piece[0].equalsIgnoreCase("READ"))
```

und

```
else if (piece[0].equalsIgnoreCase("WRITE"))
```

und

```
if (s.equalsIgnoreCase("EXIT"))
```

wird diese Funktionalität jedoch wie auch in der vergangenen Testataufgabe unterstützt. Auch die Eingaben der Dateinamen sind nicht von Groß- und Kleinschreibung abhängig, sofern eine Datei nicht neu erstellt werden muss, um in diese schreiben zu können. Existiert die zu beschreibende Datei noch nicht, so wird eine mit der entsprechenden Groß- und Kleinschreibung, wie im Befehl angegeben, angelegt. Der Zugriff erfolgt jedoch "non case sensitive". Auf der anderen Seite erfolgt das Schreiben der Daten in eine Datei immer "case sensitive".

Durch die eigene Implementierung der File-Klasse `MyFile.java` ist die Benutzereingabe so konzipiert, dass der Benutzer nur den Namen der entsprechenden Datei eingeben muss. Innerhalb des Programms wird dann automatisch mit einer Text-Datei weitergearbeitet. Dies hat zur Folge, dass die `.bak-Datei` auch die entsprechende Endung der Datei `.txt` mitübernimmt, was jedoch rein technisch erstmal kein Problem darstellt. Dies ist wichtig für den Benutzer, da, wenn er nun eine Eingabe mit der Dateiendung vornimmt, diese gedoppelt wird. Es ist vom Benutzer nut gefordert den **DATEINAMEN** anzugeben.

Für das Durchspielen der verschiedenen Beispiele werden folgende Dateien mit dem entsprechenden Inhalt vorausgesetzt:

Speiseplan.txt

```
Montag: Sushi
Dienstag: Burger
Mittwoch: Porridge
Donnerstag: Brot und Aufschnitt
Freitag: Suppe
Samstag: Käsekuchen
Sonntag: Braten
```

Zahlen.txt

```
Null
Eins
Zwei
Drei
Vier
Fünf
Sechs
Sieben
Acht
Neun
Zehn
Doppel Eins
...
```

Diese können im Ordner "*Test Files*" gefunden werden. Für individuelle Testfälle können aber natürlich auch andere Textdateien über *das Programm* oder *manuell* angelegt werden. In den folgenden Testfällen werden diese Dateien geändert. Mit den geänderten Daten wird dann auch in den folgenden Testfall weitergearbeitet.

Um die Parallelität in den eigenen Tests besser nachvollziehen zu können, schläft jeder Worker / Thread, sobald er ein Befehl ausführt für 5 Sekunden.

Alle folgenden Beispiele finden mittels **Client-Modus 1** statt:

```
Available mode:
1 - manual user input
2 - prepared automatic input
Please choose one of the modes above:
> 1
**************************************************************************
**********
ATTENTION: You have chosen mode 1!
Type a command to send:
'READ file, lineNO' OR 'WRITE file, lineNo, data' OR 'EXIT'
>
```

Dabei wurde der Server entsprechend der folgenden Ausgabe gestartet:

```
SUCCESS: Server was startet on port 5999!
SUCCESS: Worker 1 was started!
ATTENTION: Worker 1 is running...
SUCCESS: Worker 2 was started!
SUCCESS: Worker 3 was started!
ATTENTION: Worker 2 is running...
ATTENTION: Worker 3 is running...
SUCCESS: Worker 4 was started!
SUCCESS: Worker 5 was started!
ATTENTION: Worker 4 is running...
ATTENTION: Worker 5 is running...
```

In einigen Beispielen wird mit 2 parallelen Clients 1 & 2 gearbeitet. Dabei wird die Eingabe des Clients 1 einen Moment eher gestartet als die des Clients 2. Diese Verzögerung kommt durch die Tatsache zustande, dass der Mensch nur in der Lage ist Kommandos sequenziell einzugeben. :) Für eine technisch akkurate Parallelität der Eingabe sollte der automatische Client verwendet werden.

Beispiel 1: Paralleles Lesen aus einer Datei

In diesem Beispiel soll mit 2 Clients parallel aus einer Datei ausgelesen werden.

Client 1 & 2: Die Benutzereingabe/Clientausgabe für dieses Beispiel sieht wie folgt aus:

```
**************************************************************************
**********
ATTENTION: You have chosen mode 1!
Type a command to send:
'READ file, lineNO' OR 'WRITE file, lineNo, data' OR 'EXIT'
> read Speiseplan, 1
SUCCESS: Answer received: <Montag: Sushi>
**************************************************************************
**********
```

Die Serverausgabe für dieses Beispiel sieht wie folgt aus:

```
ATTENTION: Dispatcher added an element to queue. Size of queue: 1
ATTENTION: Removing an element from queue. The new size of queue is: 0
SUCCESS: Worker 1 received the job from the queue!
ATTENTION: Dispatcher added an element to queue. Size of queue: 1
ATTENTION: Removing an element from queue. The new size of queue is: 0
SUCCESS: Worker 5 received the job from the queue!
ATTENTION: Worker 1 starts reading...
ATTENTION: Worker 1 stops reading...
ATTENTION: Worker 5 starts reading...
ATTENTION: Worker 5 stops reading...
```

Auswertung

Die Serverausgabe zeigt, dass paralleles Lesen kein Problem für das Programm ist und es dementsprechend parallel von je 1 Worker pro Anfrage durchgeführt wird, denn lesende Zustände schließen sich gegenseitig nicht aus.

Beispiel 2: Paralleles Lesen aus mehreren Dateien

In diesem Beispiel soll mit 2 Clients parallel aus mehreren Dateien ausgelesen werden.

Client 1: Die Benutzereingabe/Clientausgabe für dieses Beispiel sieht wie folgt aus:

```
**************************************************************************
**********
> read speiseplan,2
SUCCESS: Answer received: <Dienstag: Burger>
**************************************************************************
**********
```

Client 2: Die Benutzereingabe/Clientausgabe für dieses Beispiel sieht wie folgt aus:

```
**************************************************************************
**********
> read Zahlen,1
SUCCESS: Answer received: <Null>
**************************************************************************
**********
```

Die Serverausgabe für dieses Beispiel sieht wie folgt aus:

```
ATTENTION: Dispatcher added an element to queue. Size of queue: 1
ATTENTION: Removing an element from queue. The new size of queue is: 0
SUCCESS: Worker 5 received the job from the queue!
ATTENTION: Worker 5 starts reading...
ATTENTION: Worker 5 stops reading...
ATTENTION: Dispatcher added an element to queue. Size of queue: 1
ATTENTION: Removing an element from queue. The new size of queue is: 0
SUCCESS: Worker 2 received the job from the queue!
ATTENTION: Worker 2 starts reading...
ATTENTION: Worker 2 stops reading...
```

Auswertung

Auch das parallele Lesen aus mehreren Dateien ist dank der separaten Worker ohne Zeitverzögerung möglich.

Beispiel 3: Paralleles Schreiben in eine Datei

In diesem Beispiel soll mit 2 Clients parallel in eine Datei geschrieben werden.

Client 1: Die Benutzereingabe/Clientausgabe für dieses Beispiel sieht wie folgt aus:

```
**********************************************************************
**********
> Write speiseplan,2,Dienstag: Salat
SUCCESS: Answer received: <Overwritten to: Dienstag: Salat>
**********************************************************************
**********
```

Client 2: Die Benutzereingabe/Clientausgabe für dieses Beispiel sieht wie folgt aus:

```
**********************************************************************
**********
> Write speiseplan,3,Mittwoch: ...mag keinen Salat, will lieber Eis!
SUCCESS: Answer received: <Overwritten to: Mittwoch: ...mag keinen Salat,
will lieber Eis!>
**********************************************************************
**********
```

Die Serverausgabe für dieses Beispiel sieht wie folgt aus:

```
ATTENTION: Dispatcher added an element to queue. Size of queue: 1
ATTENTION: Removing an element from queue. The new size of queue is: 0
SUCCESS: Worker 1 received the job from the queue!
ATTENTION: Dispatcher added an element to queue. Size of queue: 1
ATTENTION: Removing an element from queue. The new size of queue is: 0
SUCCESS: Worker 2 received the job from the queue!
ATTENTION: Worker 1 starts writing...
ATTENTION: Worker 1 stops writing...
ATTENTION: Worker 2 starts writing...
ATTENTION: Worker 2 stops writing...
```

Auswertung

Öffnet man nun die Datei `Speiseplan.txt`, so erkennt man, dass in sowohl in Zeile 2 als auch in Zeile 3 die jeweiligen Eingaben vorhanden sind. Jedoch erfolgte dies zeitlich verzögert, da die Worker beim Schreiben in die gleiche Datei sich gegenseitig ausschließen und somit der Reihenfolge nach warten mussten.

Beispiel 4: Paralleles Schreiben in mehrere Dateien

In diesem Beispiel soll mit 2 Clients parallel in mehrere Dateien geschrieben werden.

Client 1: Die Benutzereingabe/Clientausgabe für dieses Beispiel sieht wie folgt aus:

```
**************************************************************************
**********
> write zahlen,11,10
SUCCESS: Answer received: <Overwritten to: 10>
**************************************************************************
**********
```

Client 2: Die Benutzereingabe/Clientausgabe für dieses Beispiel sieht wie folgt aus:

```
**************************************************************************
**********
> write spEiSeplAn,7,Sonntag: Entenbraten
SUCCESS: Answer received: <Overwritten to: Sonntag: Entenbraten>
**************************************************************************
**********
```

Die Serverausgabe für dieses Beispiel sieht wie folgt aus:

```
ATTENTION: Dispatcher added an element to queue. Size of queue: 1
ATTENTION: Removing an element from queue. The new size of queue is: 0
SUCCESS: Worker 4 received the job from the queue!
ATTENTION: Dispatcher added an element to queue. Size of queue: 1
ATTENTION: Removing an element from queue. The new size of queue is: 0
SUCCESS: Worker 2 received the job from the queue!
ATTENTION: Worker 4 starts writing...
ATTENTION: Worker 2 starts writing...
ATTENTION: Worker 4 stops writing...
ATTENTION: Worker 2 stops writing...
```

Auswertung

Wie man an der sequenziellen Ausgabe des Servers schön sehen kann, muss Worker 2 nicht warten, bis Worker 4 mit dem Beschreiben der Datei fertig ist, da beide auf unterschiedliche Dateien zugreifen und sich somit im Sinne des Monitor- Konzeptes nicht gegenseitig ausschließen. Das Beschreiben von unterschiedlichen Dateien kann somit komplett parallel mittels mehrerer Worker stattfinden.

Beispiel 5: Paralleles Lesen und Schreiben in eine Datei

In diesem Beispiel soll mit 2 Clients parallel in mehrere Dateien geschrieben werden.

Client 1: Die Benutzereingabe/Clientausgabe für dieses Beispiel sieht wie folgt aus:

```
**********************************************************************
**********
> read zahlen,3
write zahlen,4,3
read zahlen,3
SUCCESS: Answer received: <Zwei>
**********************************************************************
**********
> SUCCESS: Answer received: <Overwritten to: 3>
**********************************************************************
**********
> SUCCESS: Answer received: <dos>
**********************************************************************
**********
```

Client 2: Die Benutzereingabe/Clientausgabe für dieses Beispiel sieht wie folgt aus:

```
**********************************************************************
**********
> read zahlen,4
write zahlen,3,dos
read zahlen,4
SUCCESS: Answer received: <Drei>
**********************************************************************
**********
> SUCCESS: Answer received: <Overwritten to: dos>
**********************************************************************
**********
> SUCCESS: Answer received: <3>
**********************************************************************
**********
```

Die Serverausgabe für dieses Beispiel sieht wie folgt aus:

```
ATTENTION: Dispatcher added an element to queue. Size of queue: 1
ATTENTION: Removing an element from queue. The new size of queue is: 0
SUCCESS: Worker 3 received the job from the queue!
ATTENTION: Worker 5 starts writing...
ATTENTION: Worker 5 stops writing...
ATTENTION: Dispatcher added an element to queue. Size of queue: 1
ATTENTION: Removing an element from queue. The new size of queue is: 0
SUCCESS: Worker 2 received the job from the queue!
ATTENTION: Worker 3 starts writing...
ATTENTION: Worker 3 stops writing...
ATTENTION: Dispatcher added an element to queue. Size of queue: 1
ATTENTION: Removing an element from queue. The new size of queue is: 0
SUCCESS: Worker 5 received the job from the queue!
ATTENTION: Worker 2 starts reading...
ATTENTION: Worker 2 stops reading...
ATTENTION: Worker 5 starts reading...
ATTENTION: Worker 5 stops reading...
```

Auswertung

Beispiel 6: Paralleles Lesen und Schreiben in mehrere Dateien

In diesem Beispiel soll mit 2 Clients aus mehreren Dateien gelesen und dabei gleichzeitig in mehrere Dateien geschrieben werden.

Client 1: Die Benutzereingabe/Clientausgabe für dieses Beispiel sieht wie folgt aus:

```
**************************************************************************
**********
> read zAhLen,1
write speiseplan,1,Montag: Nix!
SUCCESS: Answer received: <Null>
**************************************************************************
**********
> SUCCESS: Answer received: <Overwritten to: Montag: Nix!>
**************************************************************************
**********
```

Client 2: Die Benutzereingabe/Clientausgabe für dieses Beispiel sieht wie folgt aus:

```
**************************************************************************
**********
> read speiseplan,1
write zahlen,1,doppel null
SUCCESS: Answer received: <Montag: Nix!>
**************************************************************************
**********
> SUCCESS: Answer received: <Overwritten to: doppel null>
**************************************************************************
**********
```

Die Serverausgabe für dieses Beispiel sieht wie folgt aus:

```
ATTENTION: Dispatcher added an element to queue. Size of queue: 1
ATTENTION: Removing an element from queue. The new size of queue is: 0
SUCCESS: Worker 3 received the job from the queue!
ATTENTION: Worker 3 starts reading...
ATTENTION: Worker 3 stops reading...
ATTENTION: Dispatcher added an element to queue. Size of queue: 1
ATTENTION: Removing an element from queue. The new size of queue is: 0
SUCCESS: Worker 5 received the job from the queue!
ATTENTION: Dispatcher added an element to queue. Size of queue: 1
ATTENTION: Removing an element from queue. The new size of queue is: 0
SUCCESS: Worker 3 received the job from the queue!
ATTENTION: Worker 5 starts writing...
ATTENTION: Worker 5 stops writing...
ATTENTION: Worker 3 starts reading...
ATTENTION: Worker 3 stops reading...
ATTENTION: Dispatcher added an element to queue. Size of queue: 1
ATTENTION: Removing an element from queue. The new size of queue is: 0
SUCCESS: Worker 2 received the job from the queue!
ATTENTION: Worker 2 starts writing...
ATTENTION: Worker 2 stops writing...
```

Auswertung

Hier wird die Schreiberpriorität beachtet und Schreib-Zugriffe werden demnach zuerst ausgeführt, wenn Lese- und Schreib- Zugriffe gleichzeitig bzw. schnell nacheinander in verschiedener Reihenfolge angefordert wurden. Der Worker, der jedoch Lese-Zugriffe in einer anderen Datei hat, die keine Schreib-Zugriffe hat, wird von der Priorisierung der 1. Datei nicht beeinflusst, da es dort keinen Schreiber gibt, der ihn behindern könnte.

Beispiel 7: Lesen aus einer nicht vorhandenen Datei

In diesem Beispiel soll aus einer nicht vorhandenen Datei gelesen werden.

Die Benutzereingabe/Clientausgabe für dieses Beispiel sieht wie folgt aus:

```
**************************************************************************
**********
> read Fahrplan,1
SUCCESS: Answer received: <ERROR: The corresponding file does not exists!>
**************************************************************************
**********
```

Die Serverausgabe für dieses Beispiel sieht wie folgt aus:

```
ATTENTION: Dispatcher added an element to queue. Size of queue: 1
ATTENTION: Removing an element from queue. The new size of queue is: 0
SUCCESS: Worker 3 received the job from the queue!
ATTENTION: Worker 3 starts reading...
ATTENTION: Worker 3 stops reading...
```

Auswertung

Wie die Fehlermeldung zeigt, wirft das Programm dem Benutzer eine entsprechende Fehlermeldung entgegen, da man nicht aus einer Datei etwas auslesen kann, was nicht existiert. In diesem Fall wird auch keine neue Datei erstellt, da der Inhalt der Datei ja nicht bekannt wäre für das Programm. Soll aus ihr ausgelesen werden, muss sie mittels WRITE-Befehl erstellt oder aber manuell dem Ordner hinzugefügt werden.

Beispiel 8: Schreiben in eine nicht vorhandene Datei

In diesem Beispiel soll in eine noch nicht vorhandene Datei geschrieben werden.

Die Benutzereingabe/Clientausgabe für dieses Beispiel sieht wie folgt aus:

```
**************************************************************************
**********
> wrITE Notizen,1,Trink mehr!
SUCCESS: Answer received: <Overwritten to: Trink mehr!>
**************************************************************************
**********
```

Die Serverausgabe für dieses Beispiel sieht wie folgt aus:

```
ATTENTION: Dispatcher added an element to queue. Size of queue: 1
ATTENTION: Removing an element from queue. The new size of queue is: 0
SUCCESS: Worker 2 received the job from the queue!
ATTENTION: No corresponding file could be found! Creating one...
ATTENTION: Worker 2 starts writing...
ATTENTION: Worker 2 stops writing...
```

Auswertung

Die Serverausgabe zeigt, dass in diesem Fall die Datei erstellt wird, sodass die entsprechende Zeile in die Datei geschrieben werden kann. Dies ist jedoch nur der Fall, wenn die gesuchte Datei vorher in dem entsprechenden Ordner noch nicht existierte. Dabei wird in der Suche keine Rücksicht auf Groß- und Kleinschreibung genommen, wobei diese übernommen wird,

sobald die Datei neu erstellt oder neu beschrieben wird (auf Grund des "Renamings" am Ende des Schreiben-Prozesses).

Beispiel 9: Lesen einer nicht vorhandenen Zeile

In diesem Beispiel soll eine noch nicht vorhandene Zeile gelesen werden.

Die Benutzereingabe/Clientausgabe für dieses Beispiel sieht wie folgt aus:

```
**********************************************************************
**********
> REad speiseplan,10
SUCCESS: Answer received: <ERROR: READ failed - line 10 could not be found
in file>
**********************************************************************
**********
```

Die Serverausgabe für dieses Beispiel sieht wie folgt aus:

```
ATTENTION: Dispatcher added an element to queue. Size of queue: 1
ATTENTION: Removing an element from queue. The new size of queue is: 0
SUCCESS: Worker 3 received the job from the queue!
ATTENTION: Worker 3 starts reading...
ATTENTION: Worker 3 stops reading...
```

Auswertung

Wie die zurückgegebene Fehlermeldung zeigt, ist es nicht möglich aus einer noch nicht beschrieben Zeile etwas auszulesen. Ist jedoch ein Leerzeichen in dieser Zeile hinterlegt, so würde diese entsprechend ausgegebene werde. Da die Zeile jedoch "null" ist, kann sie nicht gefunden werden, da sie keinen Inhalt besitzt.

Beispiel 10: Beschreiben einer nicht vorhandene Zeile

In diesem Beispiel soll eine noch nicht vorhandene Zeile einer Datei beschrieben werden.

Die Benutzereingabe/Clientausgabe für dieses Beispiel sieht wie folgt aus:

```
**********************************************************************
**********
> WRITE zahlen,20,Twenty
SUCCESS: Answer received: <Overwritten to: Twenty>
**********************************************************************
**********
```

Die Serverausgabe für dieses Beispiel sieht wie folgt aus:

```
ATTENTION: Dispatcher added an element to queue. Size of queue: 1
ATTENTION: Removing an element from queue. The new size of queue is: 0
SUCCESS: Worker 1 received the job from the queue!
ATTENTION: Worker 1 starts writing...
ATTENTION: Worker 1 stops writing...
```

Auswertung

Die Serverausgabe zeigt, dass die Zeile erfolgreich beschrieben wurde. Ob nun diese Zeile vorher bereits existierte oder nicht ist in diesem Fall unerheblich für das Programm. Dieser Sachverhalt hat nur Auswirkungen auf den READ-Befehl.

Beispiel 11: Überschreiben einer bereits vorhandenen Zeile

In diesem Beispiel soll eine bereits vorhandene Zeile einer Datei überschrieben werden.

Die Benutzereingabe/Clientausgabe für dieses Beispiel sieht wie folgt aus:

```
**************************************************************************
**********
> write zahlen,1,
SUCCESS: Answer received: <Overwritten to: >
**************************************************************************
**********
```

Die Serverausgabe für dieses Beispiel sieht wie folgt aus:

```
ATTENTION: Dispatcher added an element to queue. Size of queue: 1
ATTENTION: Removing an element from queue. The new size of queue is: 0
SUCCESS: Worker 5 received the job from the queue!
ATTENTION: Worker 5 starts writing...
ATTENTION: Worker 5 stops writing...
```

Auswertung

Die Serverausgabe zeigt, dass die Zeile erfolgreich beschrieben wurde. Ob nun vorher dort etwas drin stand oder nicht ist in diesem Fall unerheblich für das Programm. Der vorherige Inhalt wird automatisch überschrieben.

Beispiel 12: Unvollständiger Lese-Befehl

In diesem Beispiel soll getestet werden, wie das Programm mit einem unvollständigen Lese-Befehl umgeht. In diesem Fall wird kein Dateiname mitgegeben, aus welchem gelesen werden soll.

Die Benutzereingabe/Clientausgabe für dieses Beispiel sieht wie folgt aus:

```
**************************************************************************
**********
> read ,3
SUCCESS: Answer received: <ERROR: Invalid command. Please enter a valid
filename.>
**************************************************************************
**********
```

Die Serverausgabe für dieses Beispiel sieht wie folgt aus:

```
ATTENTION: Dispatcher added an element to queue. Size of queue: 1
ATTENTION: Removing an element from queue. The new size of queue is: 0
SUCCESS: Worker 1 received the job from the queue!
```

Auswertung

Wie die Fehlerausgabe zeigt, wird dieser Befehl nicht entgegengenommen, da er nicht den Anforderungen eines gültigen Befehls entspricht. Der Benutzer bekommt dies als entsprechenden Fehler ausgegeben und bekommt sogar gesagt, welcher Parameter fehlt bzw. unvollständig ist. Danach kann er weitere Eingaben vornehmen.

Beispiel 13: Unvollständiger Schreib-Befehl

In diesem Beispiel soll getestet werden, wie das Programm mit einem unvollständigen Schreib-Befehl umgeht. In diesem Fall wird kein Dateiname mitgegeben, in welche geschrieben werden soll.

Die Benutzereingabe/Clientausgabe für dieses Beispiel sieht wie folgt aus:

```
**************************************************************************
**********
> write ,1,nö
SUCCESS: Answer received: <ERROR: Invalid command. Please enter a valid
filename.>
**************************************************************************
**********
```

Die Serverausgabe für dieses Beispiel sieht wie folgt aus:

```
ATTENTION: Dispatcher added an element to queue. Size of queue: 1
ATTENTION: Removing an element from queue. The new size of queue is: 0
SUCCESS: Worker 5 received the job from the queue!
```

Auswertung

Wie die Fehlerausgabe zeigt, wird dieser Befehl nicht entgegengenommen, da er nicht den Anforderungen eines gültigen Befehls entspricht. Der Benutzer bekommt dies als entsprechenden Fehler ausgegeben und bekommt sogar gesagt, welcher Parameter fehlt bzw. unvollständig ist. Danach kann er weitere Eingaben vornehmen.

Beispiel 14: Unzulässiger Lese-Befehl

In diesem Beispiel soll getestet werden, wie das Programm mit einem unzulässigen Lese-Befehl umgeht. In diesem Fall wird keine Integer als Zeilennummer angegeben.

Die Benutzereingabe/Clientausgabe für dieses Beispiel sieht wie folgt aus:

```
**************************************************************************
**********
> READ zahlen,eins
SUCCESS: Answer received: <ERROR: Bad line number input. Line number has to
be an integer number greater than 0.>
**************************************************************************
**********
```

Die Serverausgabe für dieses Beispiel sieht wie folgt aus:

```
ATTENTION: Dispatcher added an element to queue. Size of queue: 1
ATTENTION: Removing an element from queue. The new size of queue is: 0
SUCCESS: Worker 5 received the job from the queue!
```

Auswertung

Wie die Fehlerausgabe zeigt, wird dieser Befehl nicht entgegengenommen, da er nicht den Anforderungen eines gültigen Befehls entspricht. Der Benutzer bekommt dies als entsprechenden Fehler ausgegeben, kann danach aber weitere Eingaben vornehmen.

Beispiel 15: Unzulässiger Schreib-Befehl

In diesem Beispiel soll getestet werden, wie das Programm mit einem unzulässigen Schreib-Befehl umgeht. In diesem Fall wird keine Integer als Zeilennummer angegeben.

Die Benutzereingabe/Clientausgabe für dieses Beispiel sieht wie folgt aus:

```
**************************************************************************
**********
> WRITE zahlen,zwei,ZWEI
SUCCESS: Answer received: <ERROR: Bad line number input. Line number has to
be an integer number greater than 0.>
**************************************************************************
**********
```

Die Serverausgabe für dieses Beispiel sieht wie folgt aus:

```
ATTENTION: Dispatcher added an element to queue. Size of queue: 1
ATTENTION: Removing an element from queue. The new size of queue is: 0
SUCCESS: Worker 1 received the job from the queue!
```

Auswertung

Wie die Fehlerausgabe zeigt, wird dieser Befehl nicht entgegengenommen, da er nicht den Anforderungen eines gültigen Befehls entspricht. Der Benutzer bekommt dies als entsprechenden Fehler ausgegeben, kann danach aber weitere Eingaben vornehmen.

Beispiel 16: Lesen einer negativen Zeilennummer

In diesem Beispiel soll aus eine negative Zeilennummer gelesen werden.

Die Benutzereingabe/Clientausgabe für dieses Beispiel sieht wie folgt aus:

```
**************************************************************************
**********
> read zahlen,-10
SUCCESS: Answer received: <ERROR: Bad line number input. Please choose a
line number greater than 0.>
**************************************************************************
**********
```

Die Serverausgabe für dieses Beispiel sieht wie folgt aus:

```
ATTENTION: Dispatcher added an element to queue. Size of queue: 1
ATTENTION: Removing an element from queue. The new size of queue is: 0
SUCCESS: Worker 5 received the job from the queue!
```

Auswertung

Wie die Fehlerausgabe zeigt, wird dieser Befehl nicht entgegengenommen, da er nicht den Anforderungen eines gültigen Befehls entspricht. Der Benutzer bekommt dies als entsprechenden Fehler ausgegeben, kann danach aber weitere Eingaben vornehmen.

Beispiel 17: Beschreiben einer negativen Zeilennummer

In diesem Beispiel soll in eine negative Zeilennummer geschrieben werden.

Die Benutzereingabe/Clientausgabe für dieses Beispiel sieht wie folgt aus:

```
**************************************************************************
**********
> wRiTe zahlen,-5,Minus Fünf
SUCCESS: Answer received: <ERROR: Bad line number input. Please choose a
line number greater than 0.>
**************************************************************************
**********
```

Die Serverausgabe für dieses Beispiel sieht wie folgt aus:

```
ATTENTION: Dispatcher added an element to queue. Size of queue: 1
ATTENTION: Removing an element from queue. The new size of queue is: 0
SUCCESS: Worker 1 received the job from the queue!
```

Auswertung

Wie die Fehlerausgabe zeigt, wird dieser Befehl nicht entgegengenommen, da er nicht den Anforderungen eines gültigen Befehls entspricht. Der Benutzer bekommt dies als entsprechenden Fehler ausgegeben, kann danach aber weitere Eingaben vornehmen.

Beispiel 18: Unbekannter Befehl

In diesem Beispiel soll getestet werden, wie das Programm mit einem unbekannten Befehl umgeht.

Die Benutzereingabe/Clientausgabe für dieses Beispiel sieht wie folgt aus:

```
**************************************************************************
**********
> Delete zahlen,1
SUCCESS: Answer received: <ERROR: The given command is unknown!>
**************************************************************************
**********
```

Die Serverausgabe für dieses Beispiel sieht wie folgt aus:

```
ATTENTION: Dispatcher added an element to queue. Size of queue: 1
ATTENTION: Removing an element from queue. The new size of queue is: 0
SUCCESS: Worker 1 received the job from the queue!
```

Auswertung

An der Ausgabe erkennt man, dass das Programm auch durchaus mit falschen Befehlen klarkommt. Als der Worker sich den Auftrag aus der Job-Warteschlange abholt, so erkennt er,

dass es kein gültiger Befehl ist und gibt dem Benutzer eine entsprechende Fehlermeldung aus. Der Benutzer kann danach weitere Eingaben vornehmen und es erneut probieren.

Beispiel 19: Mehr Befehle als Worker

In diesem Beispiel soll getestet werden, was passiert, wenn man mehr Befehle als Worker hat.

Client 1: Die Benutzereingabe/Clientausgabe für dieses Beispiel sieht wie folgt aus:

```
**************************************************************************
**********
> READ zahlen,1
READ zahlen,2
READ zahlen,3
READ zahlen,4
READ zahlen,5
SUCCESS: Answer received: <doppel null>
**************************************************************************
**********
> SUCCESS: Answer received: < >
**************************************************************************
**********
> SUCCESS: Answer received: <dos>
**************************************************************************
**********
> SUCCESS: Answer received: <3>
**************************************************************************
**********
> SUCCESS: Answer received: <Vier>
**************************************************************************
**********
```

Client 2: Die Benutzereingabe/Clientausgabe für dieses Beispiel sieht wie folgt aus:

```
**************************************************************************
**********
> READ zahlen,6
READ zahlen,7
READ zahlen,8
READ zahlen,9
READ zahlen,10
SUCCESS: Answer received: <Fünf>
**************************************************************************
**********
> SUCCESS: Answer received: <Sechs>
**************************************************************************
**********
> SUCCESS: Answer received: <Sieben>
**************************************************************************
**********
> SUCCESS: Answer received: <Acht>
**************************************************************************
**********
> SUCCESS: Answer received: <Neun>
**************************************************************************
**********
```

Die Serverausgabe für dieses Beispiel sieht wie folgt aus:

```
ATTENTION: Dispatcher added an element to queue. Size of queue: 1
ATTENTION: Removing an element from queue. The new size of queue is: 0
SUCCESS: Worker 1 received the job from the queue!
ATTENTION: Worker 1 starts reading...
ATTENTION: Dispatcher added an element to queue. Size of queue: 1
ATTENTION: Removing an element from queue. The new size of queue is: 0
SUCCESS: Worker 5 received the job from the queue!
ATTENTION: Worker 5 starts reading...
ATTENTION: Worker 1 stops reading...
ATTENTION: Dispatcher added an element to queue. Size of queue: 1
ATTENTION: Removing an element from queue. The new size of queue is: 0
SUCCESS: Worker 2 received the job from the queue!
ATTENTION: Worker 2 starts reading...
ATTENTION: Worker 5 stops reading...
ATTENTION: Dispatcher added an element to queue. Size of queue: 1
ATTENTION: Removing an element from queue. The new size of queue is: 0
SUCCESS: Worker 1 received the job from the queue!
ATTENTION: Worker 1 starts reading...
ATTENTION: Worker 2 stops reading...
ATTENTION: Dispatcher added an element to queue. Size of queue: 1
ATTENTION: Removing an element from queue. The new size of queue is: 0
SUCCESS: Worker 5 received the job from the queue!
ATTENTION: Worker 5 starts reading...
ATTENTION: Worker 1 stops reading...
ATTENTION: Dispatcher added an element to queue. Size of queue: 1
ATTENTION: Removing an element from queue. The new size of queue is: 0
SUCCESS: Worker 2 received the job from the queue!
ATTENTION: Worker 2 starts reading...
ATTENTION: Worker 5 stops reading...
ATTENTION: Dispatcher added an element to queue. Size of queue: 1
ATTENTION: Removing an element from queue. The new size of queue is: 0
SUCCESS: Worker 1 received the job from the queue!
ATTENTION: Worker 1 starts reading...
ATTENTION: Worker 2 stops reading...
ATTENTION: Dispatcher added an element to queue. Size of queue: 1
ATTENTION: Removing an element from queue. The new size of queue is: 0
SUCCESS: Worker 5 received the job from the queue!
ATTENTION: Worker 5 starts reading...
ATTENTION: Worker 1 stops reading...
ATTENTION: Dispatcher added an element to queue. Size of queue: 1
ATTENTION: Removing an element from queue. The new size of queue is: 0
SUCCESS: Worker 2 received the job from the queue!
ATTENTION: Worker 2 starts reading...
ATTENTION: Worker 5 stops reading...
ATTENTION: Dispatcher added an element to queue. Size of queue: 1
ATTENTION: Removing an element from queue. The new size of queue is: 0
SUCCESS: Worker 1 received the job from the queue!
ATTENTION: Worker 1 starts reading...
ATTENTION: Worker 2 stops reading...
ATTENTION: Worker 1 stops reading...
```

Auswertung

Wie man in der Serverausgabe sieht, werden alle 5 Worker vollständig ausgelastet. Jedoch muss zwischendurch gewartet werden, wenn keine Worker mehr frei sind, um die noch offenen Befehle entgegenzunehmen. Diese bleiben dann, bis zur Entgegennahme in der Warteschlange gelagert. Sind die ersten Worker fertig, so können sie die weiteren offenen Fälle bearbeiten.

Beispiel 20: Befehl ohne gestarteten Server

In diesem Beispiel soll mit getestet werden, was passiert, wenn kein Server gestartet ist und somit der Client keine Antwort erhält.

Die Benutzereingabe/Clientausgabe für dieses Beispiel sieht wie folgt aus:

```
**************************************************************************
**********
> read speiseplan,6
ERROR: java.net.SocketTimeoutException: Receive timed out
No connection to server available. The client timed out after 300000
milliseconds and will be closed now...
```

Auswertung

Wie geplant timed der Client nach einer eingestellten Zeit von 5 Minuten aus, sollte er keine Antwort vom Server bzw. von einem Worker erhalten. Denn da durch UDP keine Connection hergestellt wird, könnte nur aufwendig überprüft werden, ob ein Server "auf der anderen Seite" läuft.

Gesamtauswertung

Die passenden Ausgaben bzw. Fehlerausgaben zeigen, dass das Programm mit allen Eventualitäten klarkommt und somit die Aufgabe entsprechend der oben genannten Anforderungen erfüllt wurde. Die Testdateien sollten am Ende des Durchlaufs wie folgt aussehen:

Speiseplan.txt

```
Montag: Nix!
Dienstag: Salat
Mittwoch: ...mag keinen Salat, will lieber Eis!
Donnerstag: Brot und Aufschnitt
Freitag: Suppe
Samstag: Käsekuchen
Sonntag: Entenbraten
```

Zahlen.txt

```
doppel null

dos
3
Vier
Fünf
Sechs
Sieben
Acht
Neun
10
Doppel Eins
...
```