Erste Überlegungen zur Umsetzung

Um die drei geforderten Funktionalitäten umzusetzen, denke ich, dass man diese in drei Methoden gießen sollte: Eine zum Lesen, eine zum Schreiben und eine zum Löschen von Dateien.

Lesen von Dateien

Um Dateien konsistent zu lesen, ohne Dateisperren zu verwenden, muss vor dem Lesen ein ZFS Snapshot erstellt werden. Der Inhalt der Datei wird dann aus dem Snapshot gelesen, anstelle des eigentlichen Dateisystems. So wird sichergestellt, dass parallele Schreibvorgänge auf der selben Datei keine Inkonsistenzen hervorrufen: Sollte ein paralleler Schreibzugriff stattfinden, wird der Snapshot aufgrund des Copy-On-Write-Prinzips immer die unveränderte Version vor dem Schreibvorgang beinhalten.

Löschen von Dateien

Das Löschen von Dateien durch die `unlink()`-Funktion oder den `rm`-Befehl sind auf Linux-Systemen atomar. Daher ergibt es keinen Sinn, vor dem Löschen einen Snapshot zu erstellen. Auftretende Konflikte müssen aber bei der Read-Write-Transaktion erkannt werden.

Lesen und Schreiben von Dateien

Hier wird zunächst eine Transaktion geöffnet, indem ein Snapshot erstellt wird. Dann wird dem Nutzer der Dateiinhalt aus dem Snapshot zur Verfügung gestellt, wie beim normalen Lesen. Nun kann der Nutzer die Datei bearbeiten, den neuen Dateiinhalt schreiben und die Transaktion "committen". Nun wird geprüft, ob der Nutzer die Datei überhaupt verändert hat. Falls nicht, wird einfach nichts getan und der Snapshot wieder entfernt. Falls doch, wird noch auf inkonsistenzen geprüft: Eine Prüfsumme wird auf der Datei aus dem Snapshot erstellt, sowie aus der Datei vom Live-System. Sollte sich die Datei geändert haben, wird ein Rollback durchgeführt. Falls nicht, wird die Datei geschrieben und der Snapshot gelöscht.

Umsetzung der Transaction-Library

Meine Transaktions-Bibliothek besteht aus zwei Klassen. Eine Klasse `ZFSUtil`, die als Interface zwischen Java und ZFS fungiert, indem sie Methoden zum Verwalten von Dateien und Snapshots bereitstellt und diese auf ZFS-Befehle abbildet.

Zweitens eine Klasse `ZFSTransaction`, die die Anforderungen der Aufgabe mithilfe der ZFSUtils-Klasse umsetzt. Sie ermöglicht es, eine Transaktion zu öffnen, innerhalb der Transaktion Dateien zu lesen, zu bearbeiten und zu löschen, sowie die Transaktion am Ende zu committen. Erst beim Commit werden tatsächlich Änderungen am Live-Dateisystem vorgenommen. Sollten Konflikte erkannt werden, wird ein Rollback durchgeführt und die Transaktion somit rückgängig gemacht.

ZFSUtil: Implementierungsdetails

Zunächst einmal hat die Klasse ZFSUtils drei Konfigurations-Variablen: `ZFS_MOUNTPOINT`, `ZFS_SNAPSHOT_DIRECTORY` und `ZFS_FILESYSTEM`. Über diese kann das zu verwaltende ZFS-Dateisystem spezifiziert werden. Standardmäßig wird ein Dateisystem `mypool/myfs` verwendet,

mit dem Standard-Mountpoint `/mypool/myfs` und dem Standart-Snapshot-Directory `/mypool/myfs/.zfs/snapshot`

Die grundlegende Funktionalität liegt in den Methoden `createSnapshot()`, `deleteSnapshot()` und `rollbackSnapshot()`. Diese verwenden die `Runtime.exec()`-Methode, um ZFS-Kommandos auf dem Hostsystem auszuführen und so ZFS-Snapshots zu verwalten. Mithilfe der `Process.waitFor()`-Methode wird jeweils gewartet, bis der erzeugte Prozess terminiert, um so eine Synchronisierung mit dem Java-Programm zu erreichen.

Zusätzlich stehen noch Methoden zum Lesen von Dateien aus dem Live-Filesystem und aus Snapshots zur Verfügung, sowie Methoden zum Schreiben und Löschen von Dateien im Live-Filesystem. Hier kommt die Java IO- bzw NIO-API zum Einsatz.

Zuletzt gibt es zum späteren Verifizieren der Konsistenz noch Methoden zum holen des File-Objekts einer Datei im Live-Filesystem bzw. in einem Snapshot, um daraus später den Zeitstempel zu extrahieren.

ZFSTransaction: Implementierungsdetails

Ein Objekt der Klasse `ZFSTransaction` hat zwei Attribute: Eine UUID, die die Transaktion eindeutig identifiziert und eine Map, in der Datei-Änderungen innerhalb der Transaktion verfolgt werden.

Mithilfe der statischen Methode `ZFSTransaction.open()` wird eine neue Transaktion geöffnet, indem ein neues ZFSTransaction-Objekt initialisiert und ihm eine zufällige UUID zugewiesen wird. Dann wird ein ZFS-Snapshot erstellt, wobei der Name des Snapshots der UUID der Transaktion entspricht. So kann die Transaktion später durch einen Rollback auf den Snapshot zurückgesetzt werden.

Nun hält das ZFSTransaction-Objekt einige Methoden bereit, um mit den Dateien zu interagieren. Zunächst gibt es die private Methode `openFile()`, die den Inhalt einer Datei aus dem Snapshot der Transaktion in die `files`-Map lädt. Ist die Datei nicht vorhanden, wird ein leerer String in die Map geschrieben, um das erstellen einer neuen, leeren Datei zu simulieren.

Folgende Methoden stehen zur direkten Interaktion bereit:

- Die `readFile()`-Methode liest den Inhalt einer Datei aus der `files`-Map aus und gibt diese zurück. Falls die Datei nicht enthalten ist, wird sie mithilfe der `openFile()`-Methode geladen.
- Die `writeFile()`-Methode schreibt den Inhalt für eine Datei in die `files`-Map. So wird der Inhalt bei Abschluss der Transaktion in die Datei geschrieben.
- Die `deleteFile()`-Methode setzt den Inhalt einer Datei in der `files`-Map auf `null`. So wird die Datei bei Abschluss der Transaktion gelöscht.
- Die `close()`-Methode schließt eine Transaktion ab, indem der entsprechende Snapshot im ZFS gelöscht wird. Zudem wird die `files`-Map geleert und die UUID auf `null` gesetzt, wodurch das ZFSTransaction-Objekt nicht weiter verwendet werden kann.
- Die `rollback()`-Methode führt einen Rollback auf den Snapshot vor der Transaktion durch. Dadurch werden alle Veränderungen seit Transaktionsbeginn verworfen. Zuletzt wird die `close()`-Methode aufgerufen.
- Die `commit()`-Methode wendet die Änderungen, die in der `files`-Map gespeichert wurden, an. Für jede Datei wird zunächst die Konsistenz geprüft, indem der Zeitstempel der Datei aus dem Snapshot mit der Datei aus dem Live-System verglichen wird. Falls die

Zeitstempel übereinstimmen, wird der neue Dateiinhalt geschrieben. Wird an irgendeiner Stelle eine Inkonsistenz (Abweichender Zeitstempel) festgestellt, so wird der Schreibvorgang abgebrochen und ein Rollback durchgeführt. Zuletzt wird die `close()`-Methode aufgerufen und die Transaktion somit geschlossen.

Entwicklung eines Brainstorming-Tools

Nun habe ich auf Basis der zuvor angelegten Bibliothek ein Brainstorming-Tool entwickelt, das es ermöglicht, Ideen zu erstellen und Kommentare zu hinterlassen. Jede Änderung wird mithilfe einer atomaren ZFS-Transaktion durchgeführt, welche im Falle einer Inkonsistenz wieder rückgängig gemacht werden kann.

Das Programm kann entweder direkt über die Kommandozeile mit entsprechenden Parametern verwendet werden, oder ohne Parameter um eine REPL zu starten.

Folgende Funktionen bzw. Kommandos stehen zur Verfügung:

- `list` listet alle Ideen auf. Dafür werden einfach alle Dateien im System aufgelistet. Hierfür wird keine Transaktion benötigt.
- `add` fügt eine neue Idee hinzu. Der Dateiname kann über ein Argument oder über eine interaktive Eingabe übergeben werden, der Inhalt (Beschreibung der Idee) wird in jedem Fall vom Standard-In gelesen. Nachfolgend wird eine Datei erstellt, und die Idee im JSON-Format abgespeichert. Dies geschieht im Rahmen einer ZFSTransaction, wodurch die Operation atomar und isoliert stattfindet.
- `remove` entfernt eine Idee: Somit wird die entsprechende Datei gelöscht. Dies findet im Rahmen einer ZFSTransaction statt.
- `show` liest eine Idee inklusive Kommentaren ein und gibt diese auf der Konsole aus. Um konsistenten Dateiinhalt zu gewährleisten, wird eine ZFSTransaction verwendet.
- `comment` ist die spannendste Funktion: Hier wird im Rahmen einer ZFSTransaction zunächst eine Idee inklusive der Kommentare gelesen und ausgegeben. Dann hat der Nutzer die Möglichkeit, einen neuen Kommentar hinzuzufügen. Schlussendlich wird der neue Dateiinhalt geschrieben und die Transaktion mit einem *commit* abgeschlossen. Sollte also in der Zwischenzeit ein anderer Kommentar geschrieben worden sein, so wird die Transaktion zurückgerollt.

Zum ausführen des Tools empfehle ich, eine Jar-Datei mit `./gradlew jar` zu generieren und diese dann mit `(sudo) java -jar ./build/libs/ZFS-Transactions-1.0-SNAPSHOT.jar` aufzurufen, da Gradle selbst auch Dinge auf dem Standard-Out ausgibt und daher die Bedienung etwas unübersichtlich ist.

Entwicklung eines Validierungs-Tools

Zur Validierung habe ich einige Szenarien implementiert und ausgewertet:

- Konkurrierende Schreibzugriffe auf die selbe Datei:
  Hier werden gleichzeitig zwei Transaktionen auf einer Datei geöffnet, die verschiedene Inhalte in die selbe Datei schreiben. Dann werden die Transaktionen nacheinander committet.
  Ergebnis: Beim commit der zweiten Transaktion wird eine Inkonsistenz erkannt und

zurückgerollt. Schlussendlich sind dann beide Transaktionen nicht durchgeführt und das Dateisystem ist wieder im ursprünglichen Zustand.

- Konkurrierende Schreibzugriffe auf verschiedene Dateien:
  Das selbe in Grün: Diesmal werden zwei verschiedene Dateien bearbeitet. Es sollte also zu keinen Problemen kommen.
  Ergebnis: Wie erwartet treten keine Konflikte auf und beide Dateien werden erfolgreich geschrieben.
- Konkurrierende Lese- und Schreibzugriffe:
  Hier führen zwei Threads parallel Lese- und Schreibzugriffe auf die selbe Datei aus. Der schreibende Thread schreibt eine zufällige Anzahl des Wortes `consistentcontent` in die Datei, während der lesende Thread mithilfe eines regulären Ausdrucks die Konsistenz des Inhalts verifiziert.
  Ergebnis: Es treten beim lesenden Thread keine Inkonsistenzen auf.
- Konkurrierende Schreibzugriffe:
  Hier habe ich 10 Threads erstellt, die jeweils auf die selbe Datei zugreifen und dort jeweils das Wort `consistentcontent` an den Dateiinhalt anfügen. Dieser Prozess wird je 10 mal wiederholt, sodass insgesamt 100 Transaktionen durchgeführt werden. So werden Inkonsistenzen und somit Rollbacks provoziert. Dabei zähle ich die Rollbacks, die gemacht werden, sowie auch die Exceptions, die bei Fehlern in den ZFS-Kommandos auftreten.
  Ergebnis: Leider scheint es Probleme beim simultanen Rollback mehrerer Snapshots zu geben. Das `zfs rollback` Kommando gibt hier immer einen Fehler zurück, sodass die Rollbacks extrem selten erfolgreich sind. Leider ist es mir auch nicht gelungen, die genaue Fehlermeldung der Kommandos zu extrahieren, sodass ich dem nicht weiter auf den Grund gehen konnte. Bei weniger Threads steigt die Wahrscheinlichkeit, dass ein Rollback provoziert wird und erfolgreich durchläuft.