Ergebnisbericht Übungsblatt 3 Betriebssysteme WiSe 24/25

Simon Haebenbrock, s4sihaeb@uni-trier.de, dc: SimonH (simon_7672)

Thema: Optimistische Nebenläufigkeit mit ZFS-Snapshots

Aufgabenstellung/ Vorgehen

Im Rahmen dieses dritten Übungsblatts habe ich ein transaktionales Dateiverwaltungssystem entwickelt, das auf einem ZFS-basierten Ansatz beruht. Ziel war es, parallele Transaktionen so zu realisieren, dass der Zustand eines Dateisystems zu Beginn jeder Transaktion über ZFS-Snapshots gesichert wird. Um diese Funktionalität zu erreichen habe ich eine Java Bibliothek implementiert, die atomare Dateioperationen ermöglicht, Konflikte über einen optimistischen Ansatz (basierend auf SHA-256-Hashvergleichen) erkennt und im Fehlerfall einen Rollback einleitet. Aufbauend auf diesem Fundament habe ich zudem ein einfaches Brainstorming-Tool entwickelt, mit dem Benutzer über die Konsole Ideen erfassen, lesen und kommentieren können. Abschließend habe ich ein Validierungstool realisiert, das unter simuliertem parallelen Zugriff auf eine gemeinsame Datei die Robustheit des Systems testet und die Konfliktanzahl ermittelt.

Architektur und Komponenten

Im Mittelpunkt steht die Klasse TransactionManager, diese koordiniert den gesamten Ablauf einer Transaktion. Ergänzt wird diese durch zwei innere Klassen: den SnapshotManager, der mittels ProcessBuilder echte ZFS-Befehle (Snapshot-Erstellung und Rollback) ausführt, und den ConflictDetector, der den SHA-256-Hashwert des Dateiinhalts vergleicht, um false-positive Konflikte, durch den reinen vergleich von Zeitstempeln, zu vermeiden.

Zur Durchführung der Dateioperationen habe ich eine separate Klasse FileOperation implementiert. Diese Klasse realisiert grundlegende Operationen wie Lesen, Schreiben und Löschen. Eine Hilfsklasse FileMetadata erfasst die Dateimetadaten (letzter Änderungszeitpunkt, Größe und Hash), die dann vom ConflictDetector zur Konflikterkennung herangezogen werden.

Dynamische Snapshot-Namensgebung

Um Namenskonflikte zu vermeiden, wird zu Beginn jeder Transaktion ein eindeutiger Snapshot-Name basierend auf dem aktuellen Zeitstempel generiert. Dieser Name wird in einer Instanzvariablen gespeichert, sodass er beim Rollback konsistent verwendet wird.

Optimistische Konflikterkennung

Anfangs wurde ein Vergleich sämtlicher Dateimetadaten realisiert. Im Verlauf der Optimierung stellte sich heraus, dass insbesondere Zeitstempeländerungen, die bei internen, transaktionalen Schreibvorgängen auftreten können, zu false-positiven Konflikten führen. Daher habe ich die Konflikterkennung dahingehend angepasst, dass ausschließlich der SHA-256-Hashwert des Dateiinhalts verglichen wird. Dies bedeutet: Solange sich der Inhalt der Datei nicht verändert, wird ein veränderter Zeitstempel ignoriert. Diese Entscheidung stellt sicher, dass nur inhaltliche, also externe Änderungen, als Konflikt gewertet werden – was den Prinzipien der Isolation und Atomizität entspricht.

Integration echter ZFS-Befehle

Da die Aufgabenstellung den Einsatz von ZFS-Snapshots vorsieht, werden in der Java-Bibliothek die Befehle zur Snapshot-Erstellung und zum Rollback mittels ProcessBuilder aufgerufen. Dabei wird der Befehl „sudo zfs snapshot os_trans_pool/os_trans_fs@<SnapshotName>“ ausgeführt. Für den Rollback wird analog „sudo zfs rollback os_trans_pool/os_trans_fs@<SnapshotName>“ aufgerufen. Der Pool und das Dateisystem wurden in der VM mit sprechenden Namen (os_trans_pool und os_trans_pool/os_trans_fs) erstellt (OS Transaktion Pool/ File System).

Um die Ausführung in einer realen Umgebung zu ermöglichen, habe ich in einer Ubuntu-VM gearbeitet, die in VirtualBox läuft. Diese VM bietet eine native Ubuntu-Umgebung, in der ZFS-Befehle direkt unterstützt werden – etwas, das auf meinem macOS System nicht möglich ist. Durch die Verwendung eines Shared Folders, in den der in IntelliJ entwickelte Code eingebunden ist, kann ich den Code und den Entwicklungsprozess in einer Umgebung ausführen, die einem nativen System sehr nahe kommt.

Prototyp eines Brainstorming-Tools

Aufbauend auf der transaktionalen Bibliothek habe ich ein einfaches, textbasiertes Brainstorming-Tool entwickelt. Dabei können Benutzer neue Ideen anlegen, bestehende Ideen lesen und kommentieren. Beim Anlegen wird in einem Ordner „ideas“ eine neue Textdatei („idea_<Zeitstempel>.txt“) erzeugt. Der TransactionManager startet eine Transaktion, registriert den initialen Zustand (leere Datei) und öffnet anschließend einen externen Editor (nano in der VM) zur Eingabe der Idee. Nach dem Speichern wird die Transaktion committet, wobei etwaige Konflikte automatisch erkannt und bei Bedarf ein Rollback eingeleitet wird. Bestehende Ideen können im Ordner aufgelistet und gelesen sowie in einer neuen Transaktion kommentiert werden. Diese Anwendung demonstriert, wie ein transaktionales Dateisystem in einem kollaborativen Szenario atomare Dateioperationen sicherstellt.

Validierungstool zur Erzeugung von Konflikten

Um die Robustheit des Systems zu evaluieren, habe ich ein Validierungstool implementiert, das mehrere gleichzeitige Transaktionen simuliert. Dabei greifen 10 Threads parallel auf eine gemeinsame Datei („shared/validation.txt“) zu und hängen zufällig generierten Text an. Durch diesen parallelen Zugriff entstehen gelegentlich Konflikte, die gezählt werden. In der Simulation ergab sich eine Konfliktanzahl von ca. 10–12 % der Gesamtoperationen, was den simulierten Erwartungen entspricht. Diese Metriken belegen, dass das transaktionale Konzept auch unter Last zuverlässig Konflikte erkennt und behandelt.

Diskussion und Fazit

Meine Implementierung zeigt, dass ein transaktionales Dateiverwaltungssystem auf Basis von ZFS-Snapshots in Java realisierbar ist. Die dynamische Namensgebung der Snapshots und der Fokus auf den Hashvergleich verhindern false-positive Konflikte. Die Entscheidung, in einer Ubuntu-VM zu arbeiten – da macOS nativ kein ZFS unterstützt – sowie die Einrichtung eines Shared Folders ermöglichen eine schnelle und pragmatische Entwicklung. Das Brainstorming-Tool demonstriert einen praxisnahen Anwendungsfall, in dem alle Dateioperationen atomar und konfliktfrei ausgeführt werden. Das Validierungstool bestätigt darüber hinaus die Robustheit des Systems, indem es unter parallelem Zugriff die erwartete Konfliktanzahl liefert.

Zusammengefasst zeigt die Umsetzung, dass die transaktionale Logik mittels ZFS-Snapshots und Konflikterkennung in Java zuverlässig integriert werden kann. Echte Systembefehle werden über einen ProcessBuilder ausgeführt, und die getroffenen Entscheidungen – die Wahl von Java, die Nutzung einer Ubuntu-VM und der Fokus auf den Hashvergleich – bilden eine solide Grundlage für zukünftige Anwendungen, welche ggf. weitreichendere Funktionalitäten fordern.

Quellen:

ZFS Cheat Sheet

ZFS-Dokumentation:
Ubuntu ZFS Documentation, verfügbar unter https://wiki.ubuntu.com/ZFS

Vorlesungsskript:
Sturm, P. – "2024W OS06 File Systems," Prof. Dr. Peter Sturm, University of Trier, 2024.

VirtualBox Dokumentation:
Oracle VM VirtualBox User Manual, Version 6.1 (oder die von dir verwendete Version), verfügbar unter https://www.virtualbox.org/manual/ch01.html

Java-Dokumentation:
Oracle Java SE Documentation – ProcessBuilder, verfügbar unter https://docs.oracle.com/en/java/javase/11/docs/api/java/lang/ProcessBuilder.html

Entwicklungshilfen:
GitHub Copilot wurde zur Generierung von Boilerplate-Code und zur Vervollständigung von Kommentaren verwendet.