

## 3 Übungsaufgabe #3: Verteilte Dateisysteme und Docker

In dieser Aufgabe soll eine vereinfachte Variante des Hadoop Distributed File System (HDFS) implementiert werden. Genau wie HDFS speichert auch das zu entwickelnde System Inhalte getrennt von den dazugehörigen Metadaten: Während *Datanode*-Server den Inhalt einzelner Datenblöcke aufbewahren, übernimmt ein zentraler *Namenode*-Server die Verwaltung von Metainformationen zu Dateien und die Zuteilung der Daten auf verschiedene Datanodes. Nach ihrer Implementierung werden die verschiedenen Systemkomponenten als Docker-Container auf der I4-Cloud ausgeführt, um einen Speicherdienst für große Datenmengen anzubieten. Zum einfachen Zugriff gilt es außerdem einen kommandozeilenbasierten Client zu implementieren.

### 3.1 Namenode-Server

In der ersten Teilaufgabe soll der Namenode-Server als Web-Service implementiert werden, der das Abfragen, Bearbeiten und Löschen von Dateimetadaten ermöglicht, ohne dass Konflikte zwischen Clients entstehen können. Außerdem koordiniert der Namenode die Verteilung von Daten auf verschiedene Datanodes.

#### 3.1.1 Grundgerüst (für alle)

Die Implementierung des Namenode erfolgt in der Klasse `mw.namenode.MWNameNode` als REST-Service. Die Basis-URL des Dienstes soll die Form `http://<server>/namenode` haben. Der HTTP-Server soll den Dienst dabei auf **Port 60998** zur Verfügung stellen.

Aufgabe:

→ Startup-Code für den JAX-RS-HTTP-Server

Hinweis:

- Die für JAX-RS benötigten Bibliotheken sind unter `/proj/i4mw/pub/aufgabe3/` bereitgestellt oder können aus Aufgabe 1 wiederverwendet werden.

#### 3.1.2 Metadaten für Dateien (für alle)

Der Namenode soll die Organisation von Dateien in einem flachen Namensraum ermöglichen. Dazu werden Objekte vom Typ `MWFileMetaData` verwendet, die den Namen einer Datei und deren Gesamtgröße speichern. Zudem soll zu jeder Datei eine erweiterbare Liste von zugehörigen Datenblöcken und deren Speicherort (d.h. Host und Port des zuständigen Datanode) geführt werden. Diese Metadaten zu einer Datei sollen per **GET**-Anfrage unter der URL `http://<server>/namenode/<dateiname>` abrufbar sein. Eine **GET**-Anfrage an die Basis-URL des Dienstes, also ohne zusätzliche Angabe eines Dateinamens, soll dagegen eine Auflistung aller verfügbaren Dateien und deren Größe zurückliefern.

Aufgaben:

→ Bereitstellung von Datenstrukturen zur Verwaltung der Metadaten  
→ Implementierung der Verarbeitung von **GET**-Anfragen

#### 3.1.3 Leases für Schreibzugriffe (für alle)

Um zu verhindern, dass mehrere Clients gleichzeitig Änderungen an einer Datei durchführen und damit einen inkonsistenten Zustand herbeiführen, muss der Namenode eine Koordinierungsmöglichkeit bereitstellen. Diese soll darin bestehen, dass der Server exklusive Schreibrechte in Form von zeitbasierten Leases vergibt. Die Datei wird dadurch für einen bestimmten Zeitraum gesperrt, nach dessen Ende die Schreibrechte automatisch verfallen. Durch Letzteres wird vermieden, dass im Fehlerfall (z. B. Absturz des Clients) die Datei permanent gesperrt bleibt.

Die Anforderung eines Lease auf eine Datei soll mittels **POST**-Anfrage auf die Datei-URL mit angehängtem Pfadsegment `lock` möglich sein. Die vollständige URL ist folglich `http://<server>/namenode/<dateiname>/lock`. Als Rückgabe liefert der Namenode eine eindeutige Lease-ID, welche dazu genutzt werden kann, den bestehenden Lease vor Ablauf des Zeitlimits zu verlängern. Andererseits kann der Client mit `unlock` aber auch eine sofortige Freigabe des Lease erwirken.

Ist bei Erhalt einer Lease-Anfrage eine Datei bereits gesperrt, soll der Server den HTTP-Fehlercode 409 zurückliefern. Falls eine Lease-Anfrage für eine Datei eintrifft, welche noch nicht existiert, soll der Namenode die Datei anlegen und das Lease herausgeben.

Aufgaben:

→ Erweiterung der Klasse `MWFileMetaData` zum Speichern des aktuellen Lease-Status  
→ Implementierung der Lease-Anfrage im HTTP-Server

Hinweise:

- Beim Abruf der Metadaten aus Teilaufgabe 3.1.2 sollen die zusätzlichen Felder für den Lease-Status nicht mit ausgegeben werden.
- Die Verwaltung des Lease-Zustands ist serverseitig ohne weitere Threads möglich.

### 3.1.4 Allokation von Datenblöcken (für alle)

Als zentrale Komponente ist der Namenode auch für die Zuteilung von Datenblöcken auf Datanodes verantwortlich. Jeder Datenblock im Gesamtsystem soll dazu durch eine eindeutige Block-ID identifizierbar sein, unter welcher der entsprechende Block von einem Datanode abgerufen werden kann. Bei Erhalt einer POST-Anfrage für eine Datei mit angehängtem Pfadsegment `alloc` vergibt der Namenode eine neue Block-ID und wählt aus den verfügbaren Datanodes einen Server aus, auf welchem der Datenblock künftig gespeichert werden soll. Die gewählte Block-ID und der Datanode werden dem Client in der Antwort mitgeteilt, haben aber *keine* Änderung an den Metadaten zur Folge.

Aufgabe:

→ Implementierung der Allokations-Anfrage auf Server-Seite

Hinweise:

- Die Liste der verfügbaren Datanodes soll dem Namenode beim Start als Parameter übergeben werden.
- Die Schnittstelle der Datanodes wird in Teilaufgabe 3.2.1 beschrieben.

### 3.1.5 Update von Metadaten (für alle)

Die Modifikation von Metadaten soll ebenfalls per POST-Anfrage auf die entsprechende Datei-Ressource erfolgen. Metadaten-Updates werden dabei durch ein angehängtes Pfadsegment `commit` gekennzeichnet. Der Client muss hierbei die Möglichkeit bekommen, sämtliche Metadaten einer Datei mit Ausnahme des Dateinamens zu erweitern bzw. zu modifizieren. Insbesondere soll es durch entsprechende Anfrageparameter möglich sein, die Dateigröße zu erhöhen und weitere Datenblöcke mitsamt dem jeweils zugehörigen Datanode anzufügen. Dabei ist zu prüfen, ob der anfragende Client auch eine passende Lease-ID vorweisen kann.

Aufgabe:

→ Implementierung von Updates für Metadaten

### 3.1.6 Löschen von Dateien (für alle)

Weiterhin soll die Möglichkeit bestehen, Dateien vom Namenode wieder zu entfernen. Beim Löschen muss sichergestellt sein, dass die Datei nicht gerade durch ein an einen Client vergebenes Lease gesperrt ist. Für das Löschen selbst ist hierzu kein Lease durch den Client anzufordern, jedoch darauf zu achten, dass dieser Vorgang mit Hilfe von Server-seitigen Vorkehrungen (z. B. Synchronisierung) nicht zu inkonsistenten Zuständen führen kann. Die Umsetzung soll dabei durch die in HTTP vorgesehene DELETE-Methode geschehen.

Aufgabe:

→ Verarbeitung von DELETE-Anfragen für Dateien

### 3.1.7 Persistierung des Zustands (optional für 5,0 ECTS)

Der Zustand des Namenodes, der alle Dateimetadaten umfasst, ist von zentraler Bedeutung für die Funktionsfähigkeit des verteilten Dateisystems. Nur mithilfe der Zuordnung zwischen Dateien und Blöcken ist es möglich, erstere später wieder abzurufen oder zu erweitern. Daher soll dafür gesorgt werden, dass der Absturz oder Neustart des Namenodes nicht zum Verlust des Zustands führt.

Hierzu muss der Namenode alle verwalteten Dateien sowie die Informationen über die zugeordneten Datenblöcke persistent in einer Datei speichern und beim Start des Diensts wieder von dort einlesen. In welchem Format die Daten gespeichert werden, ist freigestellt, es ist jedoch darauf zu achten, dass das Abspeichern des Zustands möglichst effizient erfolgt. Änderungen an einer einzelnen Datei sollen dementsprechend nicht dazu führen, dass der komplette Zustand des Namenodes erneut geschrieben wird.

Bevor die Antwort auf eine REST-Anfrage an den Client gesendet wird, muss sichergestellt werden, dass durch die Anfrage verursachte Zustandsänderungen auch tatsächlich auf die Festplatte geschrieben wurden (`FileChannel.force()`). Dies ist notwendig, da das Betriebssystem Schreibzugriffe häufig aus Performanzgründen im Arbeitsspeicher puffert, was bei einem Absturz jedoch zu Datenverlust führen kann [Folie 5.3:5].

Aufgaben:

- Implementierung einer effizienten, persistenten Speicherung des Namenode-Zustands
- Wiederherstellung des Zustands beim Start des Namenode

Hinweis:

- Es ist zu verhindern, dass nach einem Neustart des Namenodes Inkonsistenzen auftreten. Beispielsweise darf kein neues Lease für eine Datei vergeben werden, solange noch ein altes, gültiges Lease existiert.

## 3.2 MWDFS-Client für die Kommandozeile

Bei unserem verteilten Dateisystem befindet sich ein großer Teil der Logik auf Client-Seite. Um zu vermeiden, dass einzelne Server zum Flaschenhals bei der Datenübertragung werden, redet der Client direkt mit den Datanodes, wobei die Zugriffe über den Namenode koordiniert werden.

Zur Implementierung der Kommandozeile ist im Pub-Verzeichnis (`/proj/i4mw/pub/aufgabe3/`) zu dieser Aufgabe ein Skelett in der Klasse `mw.dfscient.MWDFSClient` vorgegeben. Diese Klasse nimmt Eingaben auf der Konsole entgegen und setzt sie in verschiedene Methodenaufrufe um, die es in dieser Teilaufgabe zu implementieren gilt.

### 3.2.1 Zugriff auf Datenblöcke auf Datanodes (für alle)

Zunächst ist es jedoch sinnvoll, sich Hilfsfunktionen zu definieren, um die Arbeit mit Datenblöcken auf den Datanodes zu vereinfachen. Dazu sollen die Methode `uploadBlock()` das Übertragen eines Blocks an den Datanode und `downloadBlock()` das Herunterladen eines Blocks ermöglichen. Die Implementierung des Datanode selbst ist im Pub-Verzeichnis zur Aufgabe als JAR (`mwdatanode.jar`) bereitgestellt. Dabei handelt es sich ebenfalls um einen REST-Dienst, welcher folgende Schnittstelle bietet:

- URL `http://<server>/datablock`  
GET Blöcke auflisten; gibt ein String-Array mit den Dateinamen aller Blöcke zurück
- URL `http://<server>/datablock/<blockid>`  
GET Byte-Array mit Inhalt des Blocks herunterladen  
POST Erstellen bzw. Anfügen eines Byte-Array an Block

Aufgabe:

→ Implementierung der Methoden `uploadBlock()` und `downloadBlock()` zum Transferieren von Blöcken von/zu einem Datanode [Folie 5.3:6]

Hinweise:

- Die Hauptklasse der Datanode-Implementierung ist `mw.datanode.MWDataNode`
- Alle Datenblöcke im System haben eine feste Größe von 1 MiB ( $2^{20}$  Byte), ausgenommen der jeweils letzte Block einer Datei.
- Der Datanode akzeptiert nur Block-IDs, die ausschließlich aus Klein-, Großbuchstaben, Zahlen oder Bindestrichen '-' bestehen. Ungültige Dateinamen werden mit dem HTTP-Fehlercode 400 (Bad Request) abgelehnt. Eine Überschreitung der Blockgröße führt zum HTTP-Fehlercode 413 (Request Entity Too Large).

### 3.2.2 Arbeiten mit Dateien (für alle)

Zur Arbeit mit Dateien muss der Client einen Basissatz an Befehlen bereithalten, welcher für die Übungsaufgabe aus dem Herauf- und Herunterladen sowie dem Löschen und Auflisten von Dateien besteht. Diese Befehle sollen in den entsprechend benannten Methoden der Vorgabe-Klasse (`mw.dfscient.MWDFSClient`) umgesetzt werden.

Die Methode `listFiles()` sendet dabei eine GET-Anfrage an den Namenode, um eine Auflistung aller Dateien zu erhalten.

Ein Aufruf an `downloadFile()` bezieht zunächst die Metadaten einer Datei vom Namenode, um die Zahl und den Speicherort der einzelnen Datenblöcke zu bestimmen. Danach werden die einzelnen Blöcke der Reihe nach von den Datanodes bezogen und lokal in eine Datei mit gleichem Namen gespeichert.

Bei einem Upload mittels `uploadFile()` soll zunächst ein Lease beim Namenode angefordert werden und im Hintergrund so lange erneuert werden, bis der gesamte Vorgang abgeschlossen ist. Die Eingabedatei wird in Blöcke aufgeteilt und jeder Datenblock der Reihe nach der Datei im verteilten Dateisystem angehängt. Dazu muss auf dem Namenode zunächst eine Block-ID allokiert werden, unter welcher der Datenblock auf dem entsprechenden Datanode abgelegt wird. Danach werden die Metadaten auf dem Namenode derart ergänzt, dass der Datenblock nun Teil der Datei ist. Sobald alle Datenblöcke transferiert wurden, darf nicht vergessen werden, das Lease wieder freizugeben.

Zum Löschen von Dateien auf dem Namenode sendet `removeFile()` eine entsprechende DELETE-Anfrage. Das Entfernen der zugehörigen Blöcke von den Datanodes ist nicht notwendig.

Aufgabe:

→ Implementierung der vorgegebenen Client-Methoden für die Befehle `list`, `upload`, `download` und `remove`

Hinweis:

- Die Gleichheit von lokal vorhandenen (Binär-)Dateien lässt sich mittels `shasum <dateiname>` überprüfen.

### 3.3 Replikation (optional für 5,0 ECTS)

Die Ablage der Daten erfolgt derzeit ohne Redundanz, verteilt auf eine möglicherweise große Anzahl von Datanodes. Da ein einzelner Ausfall eines Datanodes bereits dafür sorgt, dass Dateien nicht mehr lesbar sind, sollen Datenblöcke auf verschiedenen Datanodes parallel vorgehalten werden. Die Zahl der redundanten Kopien soll dabei pro Datei beim Anlegen festgelegt werden.

Damit dies möglich ist, muss auf Server-Seite die Struktur der Metadaten derart angepasst werden, dass ein Block auf mehreren Datanodes verfügbar sein kann. Zudem soll bei der Allokation eines Datenblocks für eine Datei automatisch eine dem Grad der Replikation entsprechende Menge an Datanodes ausgewählt werden.

Auf Client-Seite kann optional beim Upload angegeben werden, wie viele redundante Kopien in dem Gesamtsystem vorgehalten werden sollen. Dementsprechend wird bei einem Upload ein Datenblock mehrfach an verschiedene Datanodes gesendet. Beim Abruf eines Datenblocks soll bei einem Fehler nun nicht sofort abgebrochen werden, sondern zunächst versucht werden, die weiteren Replikate zu kontaktieren.

Aufgabe:

→ Erweiterung von Client und Server um redundante Speicherung

### 3.4 Bereitstellung des MWDFS als Docker-Images in der Cloud (optional für alle)

Der Dienst soll schließlich in Form mehrerer Docker-Container auf der OpenStack-Cloud des Lehrstuhls gehostet werden. Dazu sind zunächst Docker-Images anzufertigen, welche in der Lage sind, die Dienste auszuführen.

#### 3.4.1 Generisches Docker-Image für Java-Anwendungen

Als Grundlage dafür soll ein Image dienen, welches JAX-RS-basierte Server ausführen kann. Dazu werden ausgehend von einem blanken Ubuntu-Betriebssystem (Docker-Image `i4mw.cs.fau.de/ubuntu:noble`) die OpenJDK-11-Laufzeitumgebung installiert sowie die JAX-RS-Bibliotheken unter `/opt/mwcc/lib` eingerichtet.

Aufgabe:

→ Erstellung eines `Dockerfile` und Bau des Basis-Image

Hinweise:

- Da Docker nicht im CIP-Pool verfügbar ist, sollen die Images in einer OpenStack-VM mit dem Namen *dockervm* erstellt und später auch gehostet werden. Beim Starten sind der Instanztyp *i4.docker* und ein SSH-Key anzugeben (siehe OpenStack-Rechnerübung). Der Benutzername zum Einloggen ist `cloud`.
- Da VM-Instanzen auf der OpenStack-Cloud beim Herunterfahren die Daten verlieren, ist es empfehlenswert, die `Dockerfiles` im CIP zu sichern und die erstellten Images in die I4-Docker-Registry (Hostname: `i4mw.cs.fau.de`) zu pushen.
- Die I4-Docker-Registry akzeptiert die bei der Gruppenanmeldung per E-Mail mitgeteilten Nutzernamen und Passwörter für die OpenStack-Web-Oberfläche.
- Images sind unter dem eigenen Gruppennamen abzulegen, beispielsweise `gruppe0/datanode`.

#### 3.4.2 Container(-Images) für Namenode und Datanodes

Auf Basis dieses generischen Java-Image sollen nun zwei weitere Docker-Images erstellt werden, welche die eigentlichen Dienste anbieten. Dazu sollen die Images um die Klassen- bzw. JAR-Dateien ergänzt werden, welche den jeweiligen Dienst bereitstellen. Die Definition eines Einstiegspunktes sorgt später dafür, dass die Dienste durch den Aufruf von `docker run` automatisch aufgerufen werden.

Zum abschließenden Testen der Aufgabe werden drei Datanode-Container instanziiert sowie ein Namenode zu deren Verwaltung gestartet. Die Angabe eines Publish-Parameters soll dafür sorgen, dass später jeder Dienst jeweils unter einer anderen Portnummer außerhalb der OpenStack-Host-VM erreichbar ist.

Aufgaben:

- Zusammenstellung dienstspezifischer Images und deren Instanziierung
- Test mit Hilfe eines Clients aus dem CIP-Pool

Hinweis:

- Der Datanode und Namenode sollen ihre Daten, soweit vorhanden, in einem Volume speichern, um beim Neuinstanzieren des jeweiligen Containers keine Daten zu verlieren.
- Dienste sollten im Container unter der IP-Adresse `0.0.0.0` veröffentlicht werden. Dadurch ist keine Instanzspezifische IP-Adresse nötig; externe Zugriffe benötigen weiterhin die öffentliche VM-IP-Adresse.

**Abgabe: am 03.12.2025 in der Rechnerübung**