# 3 Übungsaufgabe #3: Verteilte Dateisysteme und Docker

In dieser Aufgabe soll eine vereinfachte Variante des Hadoop Distributed File System (HDFS) implementiert werden. Die verschiedenen Server-Komponenten werden anschließend 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 entwickeln.
Das Speichern von Inhalten soll getrennt von den Metadaten auf unterschiedlichen Server-Arten erfolgen. Während Datanodes den Inhalt einzelner Datenblöcke aufnehmen können, wird die Verwaltung von Metainformationen zu Dateien und die Zuteilung der Daten auf verschiedene Server von einem zentralen Namenode übernommen.

## 3.1 Namenode-Server

Die Implementierung des Namenode erfolgt in der Klasse `mw.namenode.MWNameNode` als REST-Service, basierend auf JAX-RS 2.0. Die Basis-URL des Dienstes soll die Form `http://<server>/namenode` haben. Bei Aufruf der `main()`-Methode soll ein HTTP-Server auf Port 60998 starten, welcher den Dienst zur Verfügung stellt.

Aufgabe:

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

Hinweise:

- JAX-RS ist nicht Bestandteil von Java SE; daher sind die Bibliotheken aus `jaxrs-ri-2.13` aus dem Pub-Verzeichnis zu dieser Aufgabe (`/proj/i4mw/pub/aufgabe3/`) einzubinden.
- Fehler auf Server-Seite sollen mit einem möglichst passenden HTTP-Status-Code beantwortet werden.

### 3.1.1 Metadaten für Dateien (für alle)

Der Namenode soll in Objekten vom Typ `MWFileMetaData` 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>` als XML-Dokument abrufbar sein.
Eine `GET`-Anfrage an die Basis-URL des Dienstes, d. h. ohne zusätzliche Angabe eines Dateinamens, soll dagegen eine Auflistung aller verfügbaren Dateien und deren Größe zurückliefern. Die Wahl der Repräsentationsart (MIME-Typ) für die Übertragung dieser Informationen ist hierbei freigestellt.

Aufgaben:

→ Entwurf von Datenstrukturen zur Verwaltung der Metadaten
→ Implementierung der Verarbeitung von `GET`-Anfragen

### 3.1.2 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 Möglichkeit zur Koordinierung bereitstellen. Diese soll darin bestehen, dass der Server exklusive Schreibrechte in Form von zeitbasierten Leases vergibt. Damit kann verhindert werden, 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 und dem Anfrageparameter `?action=lock` möglich sein. 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 `?action=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:

- Zusätzliche Felder für den Lease-Status sollen nicht mit in das XML-Dokument aus Teilaufgabe 3.1.1 serialisiert werden.
- Die Verwaltung des Lease-Zustands ist serverseitig ohne weitere Threads möglich.

### 3.1.3 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 auf eine Datei mit Anfrageparameter `?action=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.

Aufgaben:

→ Übergabe der Liste von verfügbaren Datanodes als Argumente auf der Kommandozeile
→ Implementierung der Allokations-Anfrage auf Server-Seite

### 3.1.4 Update von Metadaten (für alle)

Die Modifikation von Metadaten soll ebenfalls per `POST`-Anfrage auf die entsprechende Datei-Ressource erfolgen. Metadaten-Updates werden dabei per Anfrageparameter `?action=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

### 3.1.5 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

### 3.1.6 Verwaltung von Verzeichnisstrukturen (optional für 5,0 ECTS)

Der Namenode bietet in der derzeitigen Form lediglich einen flachen Namensraum. Zur besseren Organisation der Dateien soll daher das Anlegen einer hierarchischen Verzeichnisstruktur ermöglicht werden. Dazu muss der Namenode zwischen Dateien und Verzeichnisobjekten unterscheiden können. Ein Verzeichnisobjekt besitzt einen Namen und kann dabei sowohl Dateien als auch weitere Verzeichnisobjekte aufnehmen.
Neue Verzeichnisobjekte können durch eine `POST`-Methode auf die Pfad-URL mit Anfrageparameter `?action=mkdir` angelegt werden, sofern der unmittelbar übergeordnete Pfad existiert und ein Verzeichnis ist. Mit der `DELETE`-Methode kann ein Verzeichnis wieder entfernt werden, allerdings nur, wenn dieses bereits leer ist. Bei einer `GET`-Anfrage auf ein Verzeichnis wird eine Auflistung des unmittelbaren Inhalts zurückgeliefert.

Aufgaben:

→ Erweiterung der Verwaltungsstrukturen um Verzeichnisobjekte
→ Implementierung der REST-Schnittstelle entsprechend der Beschreibung

## 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 zu dieser Aufgabe ein Skelett in der Klasse `mw.dfsclient.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. Die Implementierung des Datanode selbst ist im Pub-Verzeichnis zur Aufgabe unter dem Package `mw.datanode` bereitgestellt. Dabei handelt es sich ebenfalls um einen REST-Dienst, welcher folgende Schnittstelle bietet:

- URL `http://<server>/datablock`

  `GET` Blöcke auflisten
  `DELETE` Alle Blöcke löschen (Datanode zurücksetzen)

- URL `http://<server>/datablock/<blockid>`

  `GET` Byte-Array mit Inhalt des Blocks herunterladen
  `POST` Erstellen bzw. Anfügen eines Byte-Array an Block
  `DELETE` Block löschen

Aufgabe:

→ Implementierung der Methoden `uploadBlock` und `downloadBlock` zum Transferieren von Blöcken von/zu einem Datanode.

### 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.dfsclient.MWDFSClient`) umgesetzt werden. Die Methode `listDirectory()` sendet dabei eine `GET`-Anfrage an den Namenode, um eine Auflistung des Verzeichnisinhalts zu erhalten (bei der 5,0-ECTS-Variante, d. h. ohne Implementierung von Teilaufgabe 3.1.6 bzw. 3.2.3, entspricht dies allen Dateien im verteilten Dateisystem).
Entsprechend sendet `removeFileOrDirectory()` eine `DELETE`-Anfrage, um ein Verzeichnis (optional für 5,0 ECTS) oder eine Datei auf dem Namenode zu löschen.
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.

Hinweis:

- Alle Datenblöcke im System haben eine feste Größe von 1 MiB ($2^{20}$ Byte), ausgenommen der jeweils letzte Block einer Datei.

Aufgabe:

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

### 3.2.3 Hierarchische Verzeichnisstrukturen (optional für 5,0 ECTS)

Um die Verwendung eines hierarchischen Namensschemas auch auf Client-Seite zu unterstützen, sollen noch die Befehle `cd` und `mkdir` unterstützt werden.
Die Methode `makeDirectory()` dient bei Verwendung des Clients dazu, entsprechende Anfragen zum Anlegen von Verzeichnissen an den Namenode zu verschicken.
Mit Hilfe der Methode `changeDirectory()` kann der Benutzer das aktuelle Arbeitsverzeichnis einstellen, welches in der Variable `curdir` verwaltet wird. Alle relativen Pfadangaben für Befehle beziehen sich auf dieses Arbeitsverzeichnis. Absolute Pfadangaben sind durch einen Schrägstrich (`/`) am Anfang gekennzeichnet. Leere Pfadbestandteile sowie die Angabe von „`.`“ beziehen sich auf die derzeitige Ebene, während „`..`“ die in der Hierarchie nächsthöhere Ebene bezeichnet. Vor dem Speichern eines Pfades in `curdir` oder der Übermittlung als URL-Bestandteil sind die Pfadangaben entsprechend zu normalisieren.

Aufgaben:

→ Erweiterung der bestehenden Befehle zur Unterstützung von relativen und absoluten Pfadangaben
→ Implementierung von `cd` und `mkdir`

## 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 (für alle)

Der Dienst soll schließlich in Form mehrerer Docker-Container auf der OpenStack-Cloud des Lehrstuhl 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 `faui42.cs.fau.de:8082/ubuntu:trusty`) die OpenJDK-7-Laufzeitumgebung installiert sowie die JAX-RS-Bibliotheken unter `/opt/mwcc/lib` eingerichtet.

Aufgabe:

→ Erstellen eines `Dockerfile` und Bauen 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 vorherige Tafelübung). Zum Einloggen per SSH den Benutzernamen `cloud` verwenden.
- Da VM-Instanzen auf der OpenStack-Cloud beim Herunterfahren die Daten verlieren, ist es empfehlenswert, die `Dockerfile`s im CIP zu sichern und die Images in die I4-Docker-Registry (Hostname und Port: `faui42.cs.fau.de:8082`) zu pushen.
- Zum Anmelden an der I4-Docker-Registry, sind Benutzername und Passwort für die OpenStack-Weboberfläche zu verwenden. Images immer unter dem eigenen Gruppennamen ablegen, z. B. `gruppe23/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:

→ Zusammenstellen dienstspezifischer Images und deren Instanziierung.
→ Testen mit Hilfe eines Clients aus dem CIP-Pool.

## Abgabe: am 07.12.2016 in der Rechnerübung