

# Middleware – Cloud Computing – Übung

## Aufgabe 3: Verteilte Dateisysteme

---

Wintersemester 2020/21

Michael Eischer, Laura Lawniczak, Tobias Distler

Friedrich-Alexander-Universität Erlangen-Nürnberg  
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)

[www4.cs.fau.de](http://www4.cs.fau.de)



Lehrstuhl für Verteilte Systeme  
und Betriebssysteme



FRIEDRICH-ALEXANDER  
UNIVERSITÄT  
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

## Verteilte Dateisysteme

- Dateisysteme

- Hadoop Distributed File System (HDFS)

## Container-Betriebssystemvirtualisierung

- Motivation

- Docker

  - Einführung

  - Architektur

  - Arbeitsablauf

  - Hinweise

## Aufgabe 3

- Übersicht

- Hinweise zu Java

## **Verteilte Dateisysteme**

---

**Dateisysteme**

## Lokale Dateisysteme

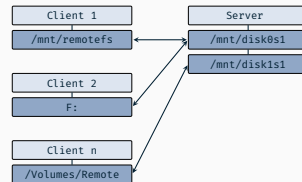
- Adressierung von Daten auf physikalischen Datenträgern
- *Beispiele:* FAT32, Ext4, Btrfs

## Netzwerk-Dateisysteme

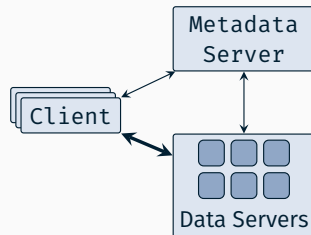
- Zugriff auf entfernte, persistente Daten über Rechnergrenzen
- *Beispiele:* Andrew File System (AFS), Network File System (NFS)

### ⚡ Probleme

- Fehleranfälligkeit (z.B. Ausfall von Netzwerkverbindungen)
- Flaschenhalsbildung durch Ungleichgewicht (Anzahl Clients vs. Server)



- Trennung von Belangen (engl. *separation of concerns*)
  - Indizierung
  - Datenverwaltung
- Replikation der Daten für höhere Ausfallsicherheit  
→ Einhaltung von *Service-Level-Agreement*, kurz: SLA
- Auflösung von Konflikten zwischen Clients
- Beispiele:
  - Ceph
  - Google File System
  - **Hadoop Distributed File System**



## **Verteilte Dateisysteme**

---

**Hadoop Distributed File System (HDFS)**

- Teil des Apache Hadoop Frameworks für skalierbare, verteilte Datenverarbeitung



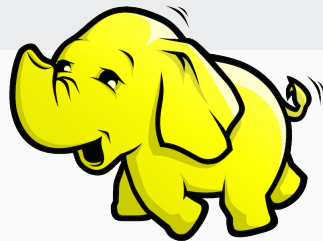
Quelle der Illustration: <https://blog.codecentric.de/2013/08/einfuehrung-in-hadoop-die-wichtigsten-komponenten-von-hadoop-teil-3-von-5/>

## Konzepte

- Write-once, read-many (WORM)
- Replikation
- Datenlokalität („rack-aware“)

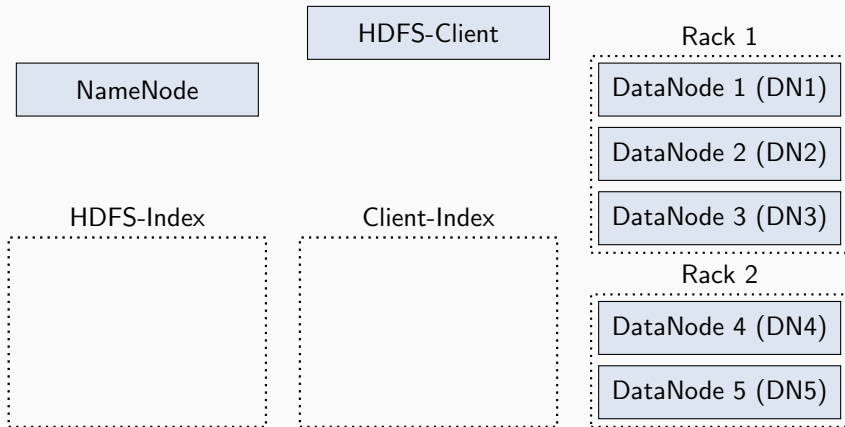
## Architektur

- HDFS-Client
- NameNode → Namensraum (Index, Metadaten)
- DataNode → Blockreplikate (Blockdaten + Metadaten)



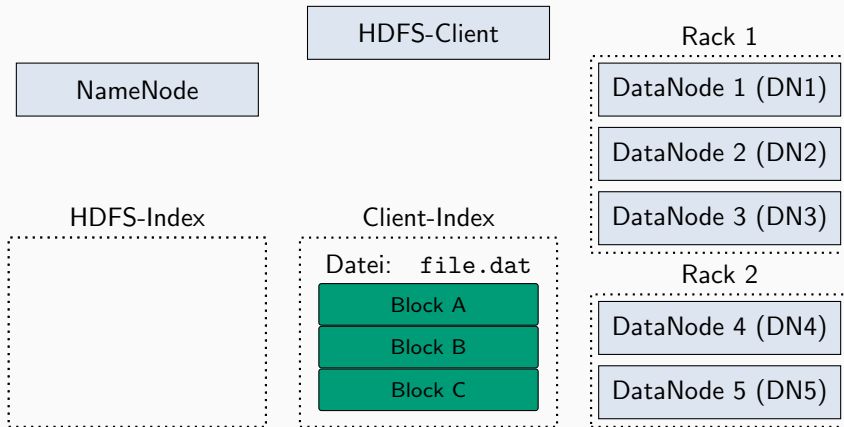


# Hadoop Distributed File System (HDFS)

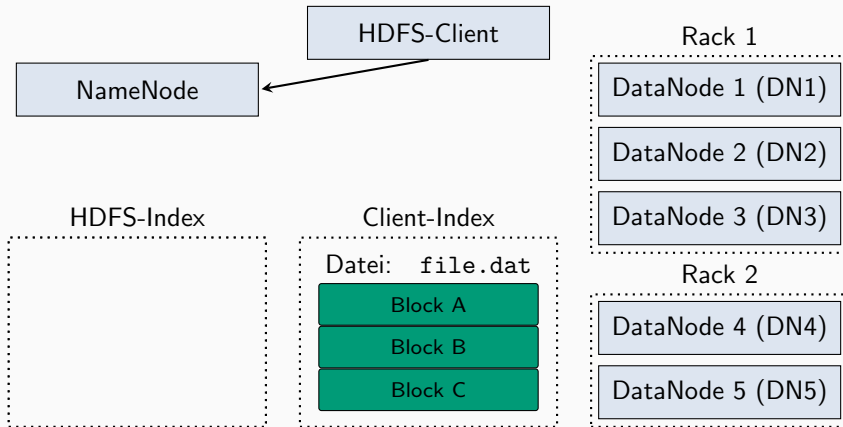


## ■ System-Konfiguration

- 1x HDFS-Client
- 1x NameNode
- 5x DataNodes (Rack 1: DN1–3, Rack 2: DN4–5)

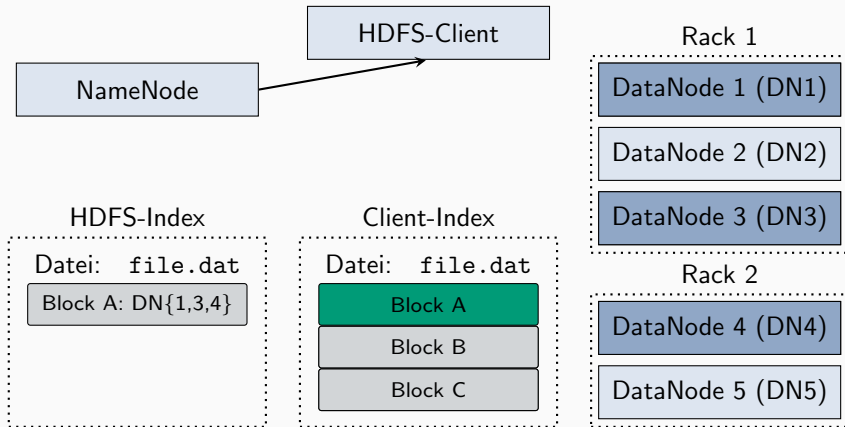


- HDFS-Client legt die aus drei Blöcken (A, B und C) bestehende Datei `file.dat` im HDFS an



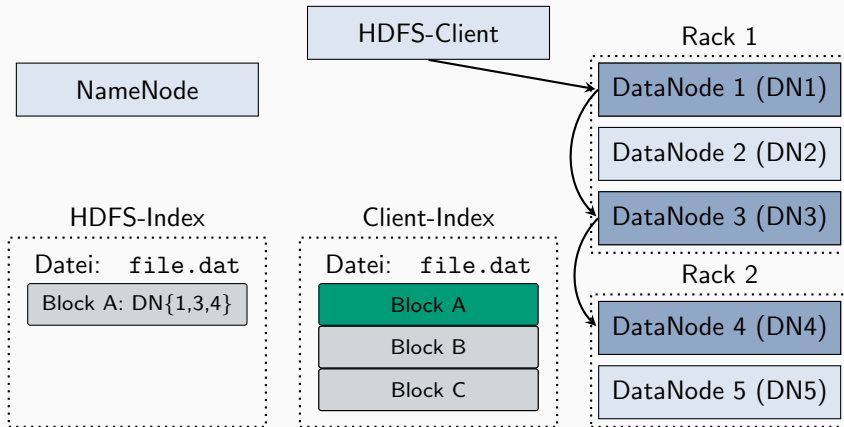
1. HDFS-Client → NameNode:

Anforderung eines sog. Lease (dt. *Miete*) für das Schreiben der Datei `file.dat`

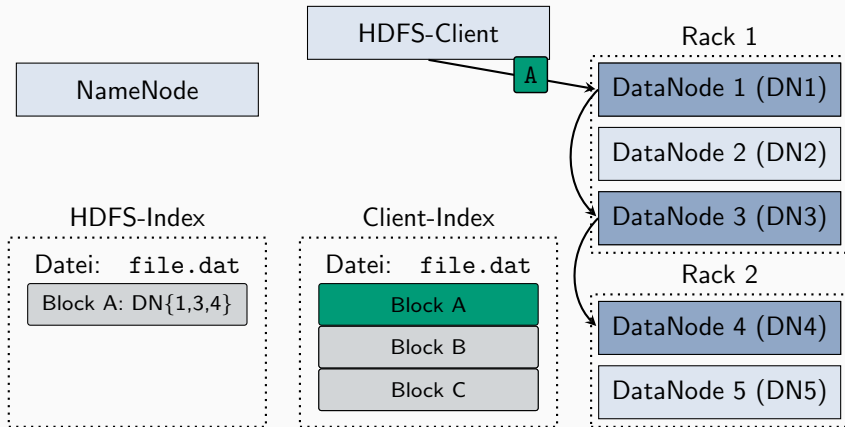


## 2. NameNode → HDFS-Client:

Erteilung des Lease, Erzeugung einer Block-ID für den ersten Block (Block A),  
Zuteilung der Replikate (DN1, DN3 und DN4)

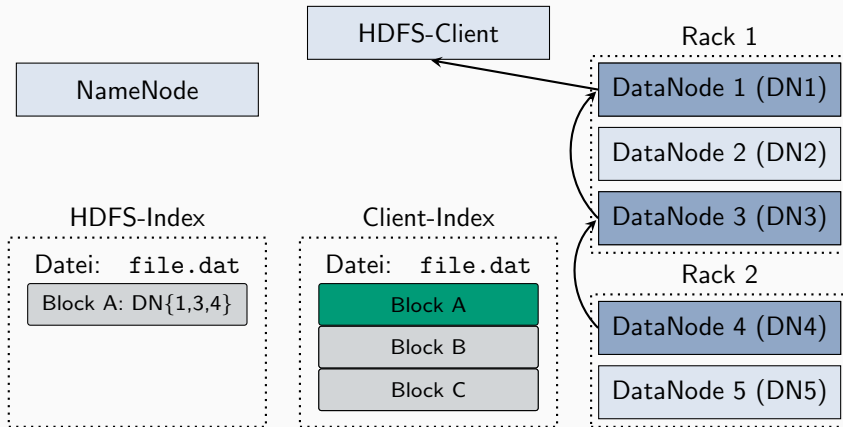


3. „Daten-Pipeline“ zur Vorbereitung der Schreiboperationen von Block A:  
HDFS-Client — DN1 — DN3 — DN4



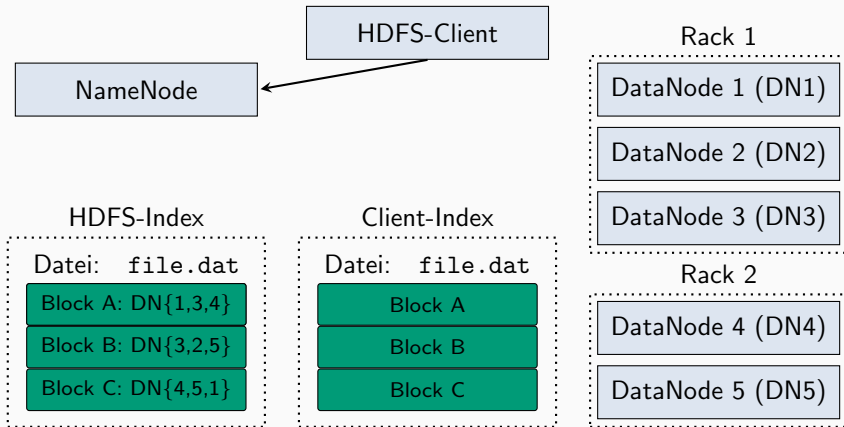
#### 4. Durchführung der Schreiboperationen:

HDFS-Client sendet Block A an DN1, DN1 sendet empfangenen Block A an DN3, DN3 sendet empfangenen Block A an DN4



##### 5. Bestätigung der Schreiboperationen:

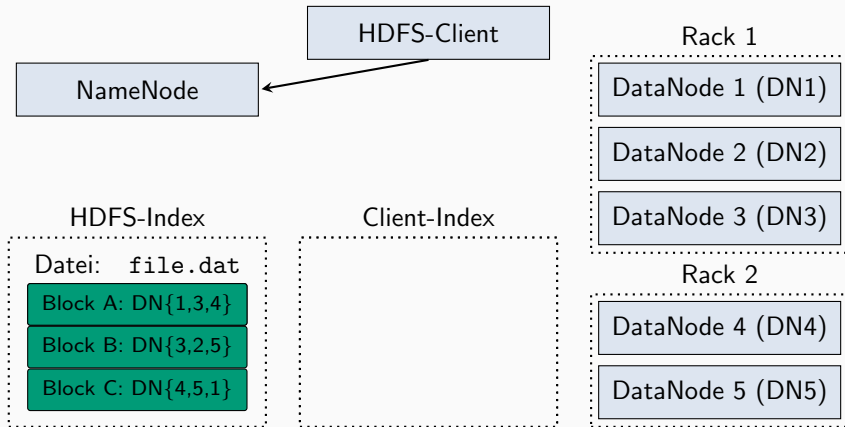
Jede DataNode bestätigt das erfolgreiche Schreiben von Block A entlang der Pipeline (Abbau)



#### 6. HDFS-Client → DataNodes:

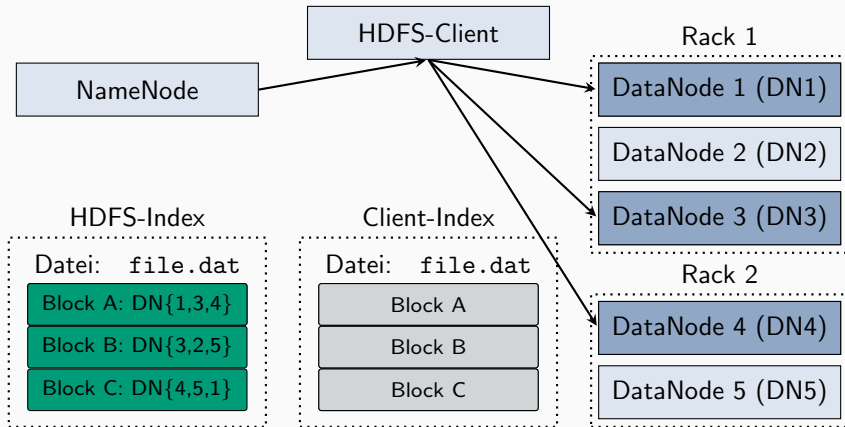
Analog werden die restlichen Blöcke der Datei vom HDFS-Client an die DataNodes verschickt;  
HDFS-Client benachrichtigt NameNode von erfolgreicher Schreiboperation





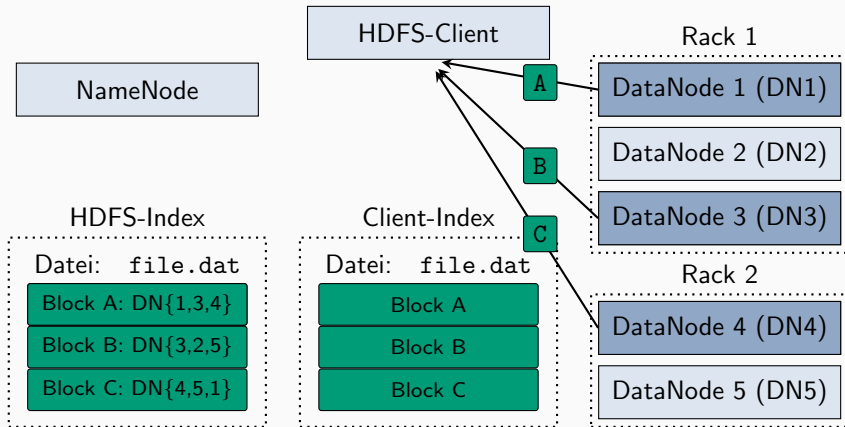
#### 1. HDFS-Client → NameNode:

Anforderung der DataNodes-Liste: Alle DataNodes, die Blöcke der zu lesenden Datei `file.dat` speichern



2. NameNode → HDFS-Client, HDFS-Client → DataNodes:

Client erhält DataNodes-Liste und wählt den ersten DataNode für jeden der Datenblöcke

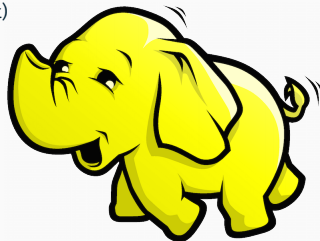


### 3. DataNodes → HDFS-Client:

HDFS-Client liest die Blöcke sequentiell, DataNodes senden die angeforderten Blöcke an den HDFS-Client

## ■ (Weitere) HDFS-Details

- Herzschlag-Nachrichten (engl. heartbeat) von DataNodes zum NameNode
  - Alle drei Sekunden (Default) ein Herzschlag
  - Replikationsfaktor sicherstellen
  - Grundlast bei sehr großen Clustern
- Block-Report: NameNode generiert Metadaten aus den Block-Reports
  - Umfangreicher Bericht über alle Blöcke alle 60 Minuten (Default)
  - Löschen ungenutzter Blöcke
- NameNode
  - *Die Sollbruchstelle des Systems?*



## ■ Literatur



Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler

### **The Hadoop distributed file system**

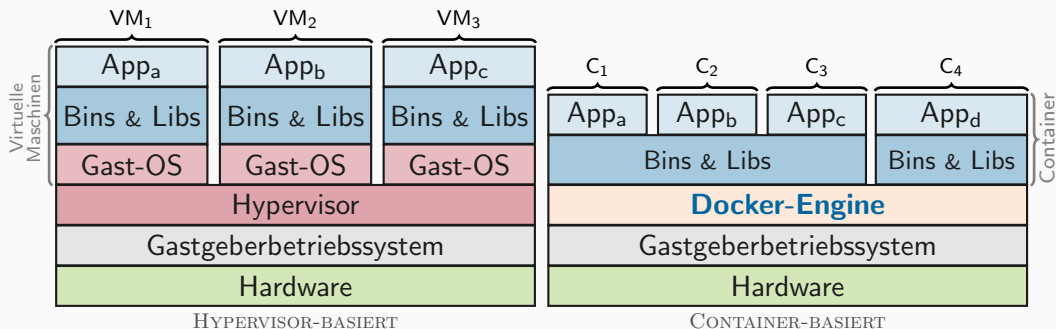
*Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST '10), pages 1–10, 2010.*

# **Container- Betriebssystemvirtualisierung**

---

## **Motivation**

# Virtualisierungsformen im Vergleich



- Hypervisor-basierte Virtualisierung
  - Erlaubt Virtualisierung von kompletten Betriebssystemen
- Container-basierte Virtualisierung
  - Leichtgewichtig: Hypervisor entfällt, kleinere Abbilder, schnelleres Hochfahren
  - Bindung an Betriebssystemkernel und vorhandene Treiber
  - Im Rahmen dieser Übung betrachtet: **Docker**



# Container- Betriebssystemvirtualisierung

---

Docker

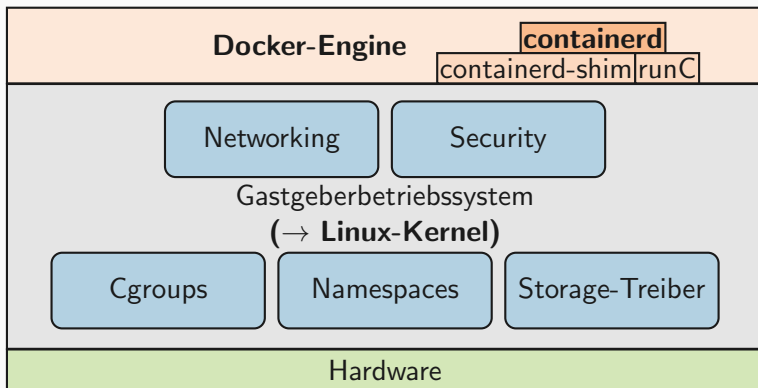


Video: „Introduction to Docker?“

Vortrag von Docker-Erfinder Solomon Hykes

(Link: <https://www.youtube.com/watch?v=Q5P0uMHxW-0>, Dauer: ~47 Min.)





- Docker setzt auf bereits existierenden Linux-Komponenten auf
  - Dominierende Komponenten
    - Ressourcenverwaltung: Control Groups
    - Namensräume
    - Überlagerte Dateisysteme
- } **containerd & runC**

- Control Groups (cgroups) ermöglichen das Steuern und Analysieren des Ressourcenverbrauchs bestimmter Benutzer und Prozesse
- Durch Control Groups abgedeckte Ressourcen
  - Speicher (RAM, Swap-Speicher)
  - CPU
  - Disk-I/O
- Funktionsweise
  - cgroups-Dateisystem mit Pseudoverzeichnissen und -dateien
  - Prozesse werden mittels Schreiben ihrer PID in passende Kontrolldatei zu einer Control Group hinzugefügt
  - Auflösen einer Control Group entspricht dem Entfernen des korrespondierenden Pseudoverzeichnisses



Tejun Heo

### **Control Group v2**

<https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html>, 2015.

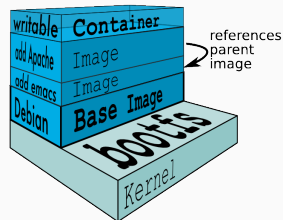
- Namensräume werden zur Isolation von Anwendungen auf unterschiedlichen Ebenen herangezogen
- **Dateisysteme**
  - Jedes Dateisystem benötigt eigenen Einhängpunkt, welcher einen neuen Namensraum aufspannt
  - Überlagerte Dateisysteme (mit Docker verwendbar: `overlayfs`) erlauben Verschmelzen von Verzeichnissen aus eigenständigen Dateisystemen
- **Prozesse**
  - Hierarchische Struktur mit einem PID-Namensraum pro Ebene
  - Pro PID-Namensraum eigener `init`-ähnlicher Wurzelprozess
  - Isolation: Prozesse können keinen Einfluss auf andere Prozesse in unterschiedlichen Namensräumen nehmen
- **Netzwerke**
  - Eigene Netzwerk-Interfaces zwischen Host und einzelnen Containern
  - Jeweils eigene Routing-Tabellen und iptables-Ketten/Regeln

## ■ Unterscheidung

- Docker-Abbild: Software-Basis zum Instanzieren von Docker-Containern
- Docker-Container: Instanziiertes Docker-Abbild in Ausführung

## ■ Inhalt eines Docker-Containers

- Dateisystem
- Laufzeitumgebung
- Binärdateien
- Systembibliotheken



Quelle der Illustration: <https://docs.docker.com/terms/layer/>

## ■ Dockerizing: „Verfrachten“ einer Anwendung in einen Container

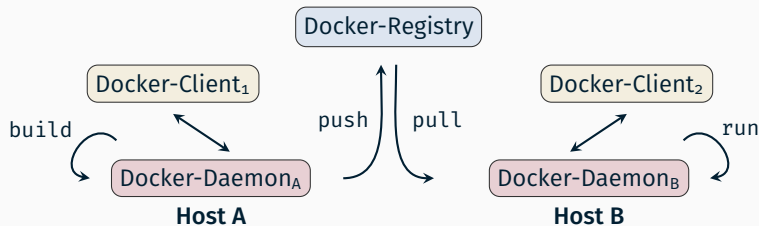
- Instanzieren eines Containers erfolgt über das Aufrufen einer darin befindlichen Anwendung
- Container an interne Anwendungsprozesse gebunden  
→ Sobald letzte Anwendung terminiert ist, beendet sich auch die Container-Instanz

## ■ Git-orientierter Arbeitsablauf

- Ähnliche Befehlsstruktur (z. B. pull, commit, push)
- Git Hub  $\Leftrightarrow$  Docker Hub

## ■ Typischer Arbeitsablauf

1. Docker-Abbilder bauen (build)
2. Ausliefern: Abbilder in Registry ein- und auschecken (push/pull)
3. Docker-Container instanzieren und zur Ausführung bringen (run)



- Von Docker, Inc. bereitgestellte Registry: **Docker Hub**
  - Cloud-Service zur Verwaltung von Docker-Abbildern bzw. -Anwendungen
  - Registrieren bzw. Anlegen eines Benutzerkontos zum Hochladen notwendig
  - Anzahl kostenloser, **öffentlicher** Repositories nicht begrenzt
  - Nur ein privates Repository kostenlos
- **Private Registry** (hier: I4-Docker-Registry)
  - Ermöglicht das Verwalten garantiert nicht-öffentlicher Repositories
  - Unabhängigkeit von Verfügbarkeit einer öffentlichen Registry
- Authentifizierung gegenüber der (privaten) Docker-Registry
  - An-/Abmelden an/von (optional spezifiziertem) Docker-Registry-Server

```
> docker login [<OPTIONS>] [<REGISTRY-HOSTNAME>]  
> [...] // Registry-zugreifende Befehle ausführen, siehe nächste Folie  
> docker logout [<REGISTRY-HOSTNAME>]
```

- **Achtung:** Weglassen eines Registry-Hostname impliziert Verwendung der **Docker-Hub**-Registry bei nachfolgenden push- oder pull-Befehlen.  
↳ (I4-Docker-Registry-Hostname: i4mw.cs.fau.de)

## ■ Vorgefertigtes Abbild aus Repository auschecken

```
> docker image pull <NAME>[:<TAG>]
```

**Hinweis:** TAG nur optional, wenn Image mit Default-Tag (= latest) existiert.

## ■ Container starten (mehr ab Folie 14)

```
> docker run -it <NAME>[:<TAG>] <COMMAND>
```

## ■ Änderungen im Container vornehmen

- Ausführen beliebiger Programme im Container mit `/bin/bash` als `COMMAND`
- Installieren von Programmen via Paket-Manager (z.B. `apt-get -yq install vim`)

- Falls Änderungen erfolgt sind und erhalten bleiben sollen
  1. Änderungen persistent machen und Abbild (lokal!) erzeugen

```
> docker commit <CONTAINER-ID> <NAME>[:<TAG>]
```

2. Abbild publizieren bzw. in Registry einspielen

```
> docker image push <NAME>[:<TAG>]
```

**Hinweis:** Da `pull` und `push` keinen Registry-Hostname vorsehen, müssen die Abbilder bei eigenen Registries über den `<NAME>`-Parameter passend gekennzeichnet sein:

- `<NAME>` besteht aus {Abbild,Benutzer}name und Registry-Hostname
- Beispiel: `$ docker image push i4mw.cs.fau.de/user/myimage:test`



## ■ In der Praxis: **Dockerfiles**

- Rezepte zum skriptbasierten Bauen eines Abbilds
- Zeilenweises Abarbeiten der darin befindlichen Instruktionen

## ■ Vordefinierte, voneinander unabhängige **Docker-Instruktionen**

- `FROM <IMAGE>[:<TAG>]`  $\mapsto$  Basisabbild auswählen (obligatorisch)
- `EXPOSE <PORT> [<PORT>...]`  $\mapsto$  Container-übergreifende Port-Freigabe
- `RUN <COMMAND>`  $\mapsto$  Ausführen eines Befehls (in *Shell-Form*)
- `COPY <SRCs> <DST>`  $\mapsto$  Dateien/Verz. ins Container-Dateisystem kopieren
- `ENTRYPOINT [<EXE>, <PARAM-1>, ...]`  $\mapsto$  Container-Einstiegspunkt setzen
  - Nur ein Einstiegspunkt (= Befehl) pro Container möglich
  - Container-Aufruf führt zwangsläufig zu Aufruf des entsprechenden Befehls
  - Parameter des letzten CMD-Befehls werden als zusätzliche Parameter an ENTRYPOINT-Aufruf angehängt, solange der Container ohne Kommando bzw. Argumente gestartet wird:  
`CMD [<EXTRA-PARAM-1>, <EXTRA-PARAM-2>, ...]`

$\rightarrow$  Vollständige Referenz: <https://docs.docker.com/reference/builder/>

## ■ Vorgehen

- Datei Dockerfile anlegen und mit Docker-Instruktionen befüllen
- Build-Prozess starten mit Kontext unter PATH, URL oder stdin (-)

```
> docker image build -t <NAME>[:<TAG>] <PATH | URL | - >
```

## ■ Beispiel-Dockerfile (Anm.: mwqueue.jar liegt im selben Verzeichnis wie das Dockerfile)

```
1 FROM      i4mw.cs.fau.de/gruppe0/javaimage
2 EXPOSE    18084
3 RUN       useradd -m -g users -s /bin/bash mwcc
4 WORKDIR   /opt/mwcc
5 RUN       mkdir logdir && chown mwcc:users logdir
6 COPY      mwqueue.jar /opt/mwcc/
7 USER      mwcc
8 ENTRYPOINT ["java", "-cp", "mwqueue.jar:lib/*", "mw.queue.MWQueueServer"]
9 CMD       ["-logdir", "logdir"]
```

- |  |                                 |
|--|---------------------------------|
| 1. Eigenes Abbild javaimage als Ausgangsbasis heranziehen                          | 6. JAR-Datei hineinkopieren     |
| 2. Port 18084 freigeben  | 7. Ausführenden Benutzer setzen |
| 3. Benutzer mwcc erstellen, diesen zur Gruppe users hinzufügen und Shell setzen    | 8. Einstiegspunkt setzen        |
| 4. Basisverzeichnis setzen (/opt/mwcc und lib-Unterverzeichnis existieren bereits) | 9. Standardargumente setzen     |
| 5. Log-Verzeichnis erstellen und Benutzerrechte setzen                             |                                 |

## ■ Besonderheiten von Docker-Abbildern

- Jeder Befehl im Dockerfile erzeugt ein neues Zwischenabbild
- Basis- und Zwischenabbilder können gestapelt werden
- Differenzbildung erlaubt Wiederverwendung zur Platz- und Zeitersparnis

## ■ Lokal vorliegende Docker-Abbilder anzeigen (inkl. Image-IDs):

```
> docker image ls
```

| REPOSITORY            | TAG    | IMAGE ID     | CREATED     | VIRTUAL SIZE |
|-----------------------|--------|--------------|-------------|--------------|
| <none>                | <none> | 7fd98daef919 | 2 days ago  | 369.8 MB     |
| i4mw.cs.fau.de/ubuntu | latest | 5506de2b643b | 11 days ago | 197.8 MB     |

- Repository: Zum Gruppieren verwandter Abbilder
- Tag: Zur Unterscheidung und Versionierung verwandter Abbilder
- Image-ID: Zur Adressierung eines Abbilds bei weiteren Befehlen

**Hinweis:** Beim Erstellen eines Abbilds mit bereits existierendem Tag wird das alte Abbild nicht gelöscht, sondern mit <none>-Tag versehen aufgehoben (siehe 1. Eintrag in Ausgabe).

## ■ Nur lokale Abbilder können über die Kommandozeile gelöscht werden

```
> docker image rm [<OPTIONS>] <IMAGE> [<IMAGE>...] # IMAGE := z. B. Image-ID
```

- Docker-Container im Hintergrund mittels -d(etached)-Flag starten

```
> docker run -d [<OPTIONS>] <IMAGE> [<COMMAND> + [ARG...]]
```

- Laufende Container und insbesondere deren **Container-IDs** anzeigen

```
> docker ps -a
```

| CONTAINER ID | IMAGE          | COMMAND               | CREATED        | ... |
|--------------|----------------|-----------------------|----------------|-----|
| ba554f163f63 | eg_pgql:latest | "bash"                | 33 seconds ago | ... |
| 345b60f9a4c5 | eg_pgql:latest | "/usr/lib/postgresql" | 7 minutes ago  | ... |
| 5496bd5d89d9 | debian:latest  | "bash"                | 46 hours ago   | ... |

| ... | STATUS                  | PORTS                   | NAMES           |
|-----|-------------------------|-------------------------|-----------------|
| ... | Up 32 seconds           | 5432/tcp                | sad_lumiere     |
| ... | Up 7 minutes            | 0.0.0.0:49155->5432/tcp | pg_test         |
| ... | Exited (0) 46 hours ago |                         | hungry_brattain |

→ -a-Flag, um auch beendete Container und deren Exit-Status anzuzeigen

- Weitere Operationen auf Containern

- Entfernen/Beenden ↪ `docker rm [OPTIONS] <CONTAINER-IDs...>`
- Attach ↪ `docker attach --sig-proxy=false <CONTAINER-IDs...>`

**Hinweis:** --sig-proxy=false nötig, um mit Ctrl-c detachen zu können

## ■ Möglichkeiten der Container-Analyse

- Logs ( $\hat{=}$  Ausgaben auf `stderr` und `stdout`) eines Containers anzeigen

```
> docker logs [<OPTIONS>] <CONTAINER-ID>
```

- Container-Metainformationen (Konfiguration, Zustand, ...) anzeigen

```
> docker inspect <CONTAINER-ID>
```

- Laufende Prozesse innerhalb eines Containers auflisten

```
> docker top <CONTAINER-ID>
```

- Jegliche Veränderungen am Container-Dateisystem anzeigen

```
> docker diff <CONTAINER-ID>
```

## ■ Es existieren eine Reihe von Container-Zuständen bzw. -Events

- Start/Wiederanlauf: `create`, `start`, `restart`, `unpause`
- Stopp/Unterbrechung: `destroy`, `die`, `kill`, `pause`, `stop`
- Anzeigen aller Event am Docker-Server  $\mapsto$  `docker events`

## ■ Nachträglich Befehle ausführen (z. B. zu Debugging-Zwecken)

- Weiteren Befehl innerhalb eines bereits laufenden Containers starten

```
> docker exec <CONTAINER-ID> <COMMAND>
```

- Eine Shell innerhalb eines bereits gestarteten Containers starten

```
> docker exec -it <CONTAINER-ID> /bin/bash
```

## ■ Netzwerk-Ports (Publish-Parameter)

- Jeder Container besitzt eigenes, internes Netzwerk
- EXPOSE-Instruktion im Dockerfile dient zu Dokumentationszwecken
- Für Zugriff von außen, interne Ports explizit auf die des Host abbilden
  - Manuell, um Host- und Container-Port exakt festzulegen

```
> docker run -p <HOST-PORT>:<CONTAINER-PORT> ...
```

- Automatisch: zufällig gewählter Port (Bereich: 49153–65535) auf Host-Seite

```
> docker run -P ...
```

- Daten innerhalb eines Containers sind an dessen Lebensdauer gebunden
- Erhalten von Daten über die Container-Lebensdauer hinweg mit Docker-Volumes
- Befehlsübersicht

- Volume erstellen

```
> docker volume create <VOLUME-NAME>
```

- Volumes auflisten

```
> docker volume ls
```

- Volume löschen

```
> docker volume rm <VOLUME-NAME>
```

- Neuen Container mit einem existierenden Volume starten

```
> docker run --mount source=<VOLUME-NAME>,target=<MOUNT-POINT> ...
```

**Hinweis:** Beim Einhängepunkt (<MOUNT-POINT>) ist darauf zu achten, dass der Benutzer im Container Schreibrechte auf das korrespondierende Verzeichnis hat.

- **Hilfsskripte** liegen in OpenStack-VM bereit unter `/usr/local/bin`
- **Verfügbare Skripte**

- Löschen aller {gestoppten,ungetaggtten} Docker-Container

```
> docker-rm-{stopped,untagged}
```

- Alle Container stoppen und Docker-Daemon neustarten

```
> docker-full-reset
```

- Alle getaggtten Abbilder in die I4-Docker-Registry hochladen

```
> docker-images-push
```

- I4-Docker-Registry durchsuchen

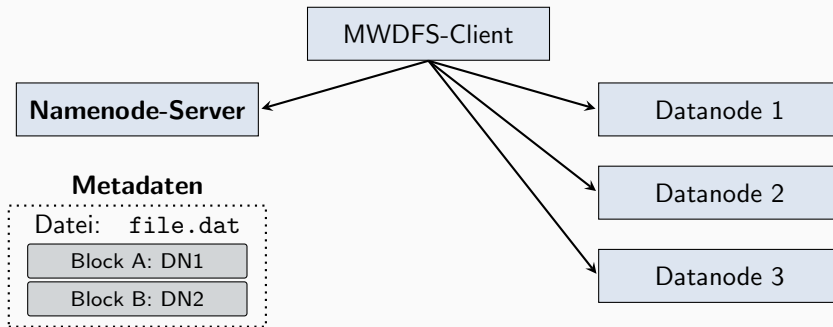
```
> docker-registry-search <SEARCH_STRING>
```



## Aufgabe 3

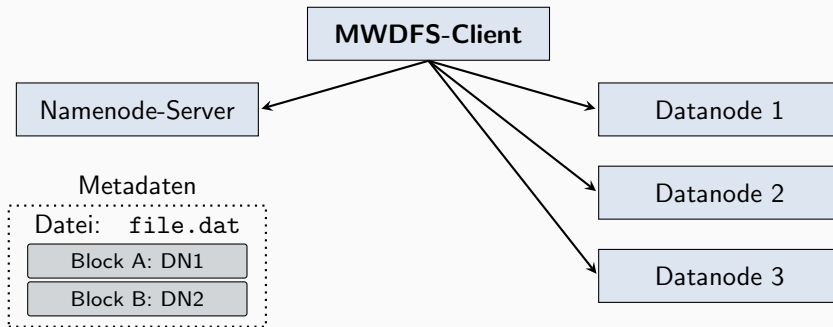
---

### Übersicht



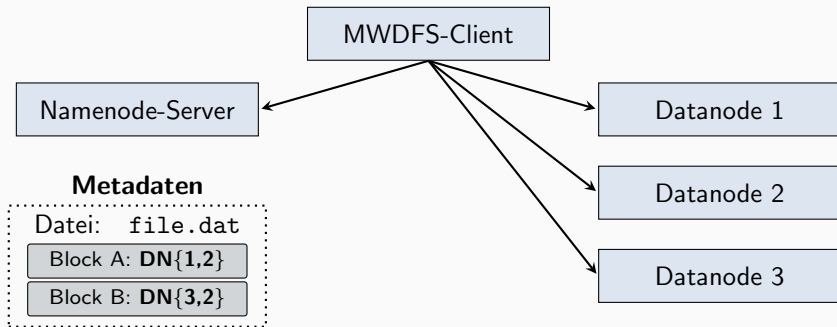
## ■ Namenode-Server

- **Metadaten**
- Leases für Schreibzugriffe
- Datei-Operationen (Anlegen, Auflisten, Löschen)

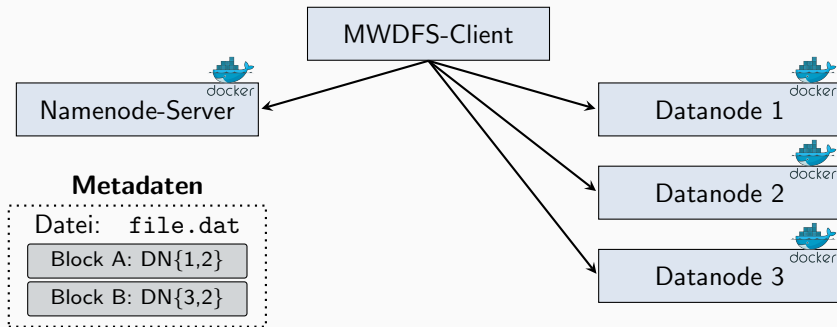


#### ■ MWDFS-Client

- Datenzugriff
- Datei-Operationen (Anlegen, Auflisten, Löschen)



- **Replikation** (optional für 5,0 ECTS)
  - Datenblöcke redundant auf mehreren Datanodes speichern
  - Erweiterung der serverseitigen **Metadaten**
- **Zustandspersistierung** (optional für 5,0 ECTS)
  - Effizientes Schreiben der Dateimetadaten bzw. Operationen
  - Wiederherstellung des Zustands nach Namenode-{Absturz,Neustart}



## ■ Docker und OpenStack

- Docker-Images erstellen
- Betrieb von Namenode-Server und drei Datanodes als **Docker-Container**
  - OpenStack-Cloud
- Zugriff auf das System über MWDFS-Client
  - CIP-Pool

## Aufgabe 3

---

Hinweise zu Java

- {S,Des}erialisierung mittels {Data,Buffered,File}{Output,Input}Stream
- Öffnen der Ströme zum Schreiben und Lesen

```
// Holen der Ausgabestroeme (Schreiben in Datei 'journal')
FileOutputStream fos = new FileOutputStream("journal");
DataOutputStream dos = new DataOutputStream(new BufferedOutputStream(fos));

// Holen der Eingabestroeme (Lesen aus Datei 'journal')
FileInputStream fis = new FileInputStream("journal");
DataInputStream dis = new DataInputStream(new BufferedInputStream(fis));
```

- Schreiben und Lesen von Daten
  - write- und read-Methoden für unterschiedliche Datentypen (z.B. writeInt(), writeBytes())
  - Erzwingen des Schreibvorgangs auf Datenträger mittels Aufruf von force() am FileChannel-Objekt  
→ boolean-Parameter von force: 'true' := Dateiinhalte **und** -metadaten schreiben

```
dos.writeLong(42);
dos.flush(); // Puffer leeren
fos.getChannel().force(true);
```

- Datanodes empfangen (POST) und senden (GET) Blockdaten als Binärdaten
- Client-Zugriffe zum Senden und Empfangen eines Datenblocks

- Für POST-Anfrage Entity-Objekt mit geeignetem MIME-Type wählen:  
„application/octet-stream“ → `MediaType.APPLICATION_OCTET_STREAM`

```
// WebTarget datanode zeigt auf http://<server>/datablock/<blockid>
public void sendBlockToDatanode(byte[] block, WebTarget datanode) {
    try {
        Response r = datanode.request()
                               .post(Entity.entity(block, MediaType.APPLICATION_OCTET_STREAM));
    } [...] // Fehlerbehandlung
}
```

- Für GET-Anfrage Response-Type auf `byte[]` setzen

```
public byte[] receiveBlockFromDatanode(WebTarget datanode) {
    byte[] block = null;
    try {
        block = datanode.request().get(byte[].class);
    } [...] // Fehlerbehandlung
    return block;
}
```