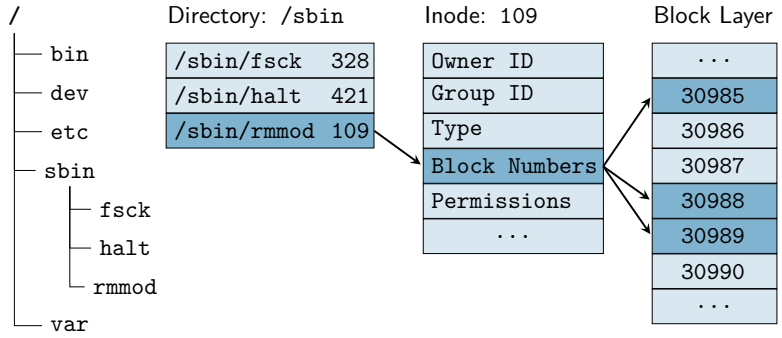
Überblick

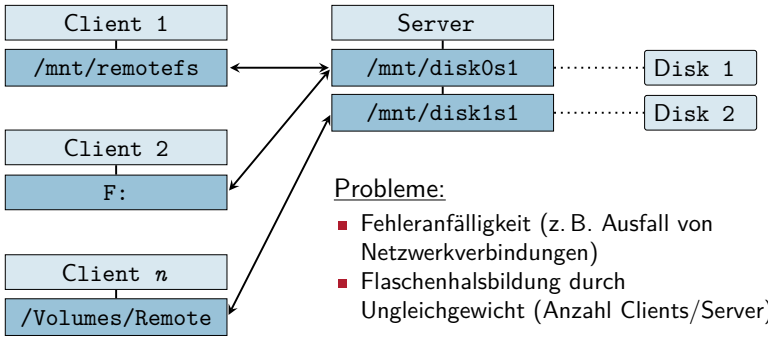
Dateisysteme

- Lokale Dateisysteme
  - Logische Schnittstelle des Betriebssystems für Zugriff auf persistente Daten durch Anwendungen und Benutzer
  - Adressierung von Daten auf physikalischen Datenträgern
  - Beispiele: `FAT32`, `Ext4`, `Btrfs`



Dateisysteme

- Netzwerk-Dateisysteme
  - Zugriff auf entfernte, persistente Daten über Rechnergrenzen hinweg
  - Für gewöhnlich werden Netzwerk-Dateisysteme in den Namensraum lokaler Dateisysteme eingebunden
  - Beispiele: Andrew File System (AFS), Network File System (NFS), Samba



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

Dateisysteme

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

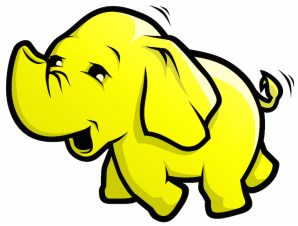
Apache Hadoop: Überblick

- Framework für skalierbare, verteilte Datenverarbeitung
  - Basiskomponenten: Hadoop Distributed File System, Hadoop MapReduce
  - Zusätzliche Komponenten (Auszug): ZooKeeper

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

Hadoop Distributed File System (HDFS)

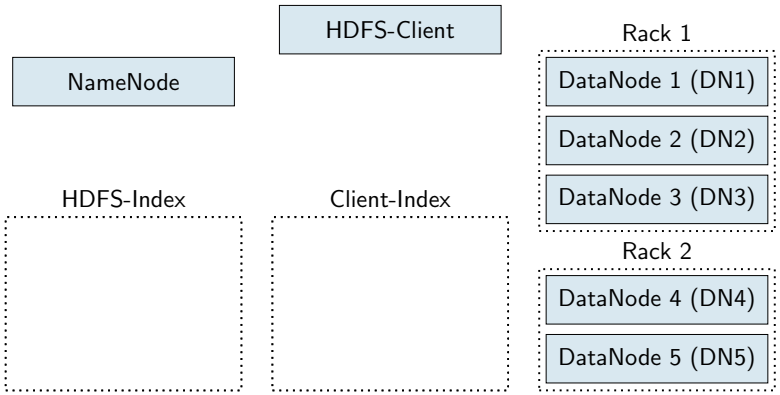
- Architektur
  - HDFS-Client
  - NameNode → Namensraum (Index, Metadaten)
  - DataNode → Blockreplikate (Blockdaten + Metadaten)
- Konzepte
  - Write-once, read-many (WORM)
  - Replikation
  - Datenlokalität („rack-aware“)



- Literatur

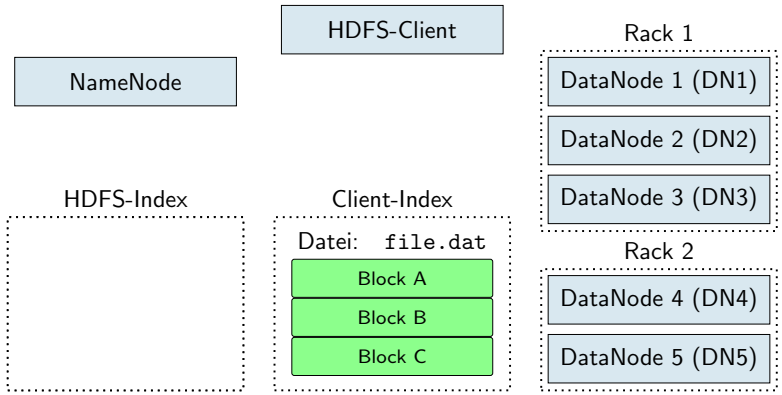
Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler
The Hadoop distributed file system
Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST '10), pages 1–10, 2010.

Hadoop Distributed File System (HDFS)



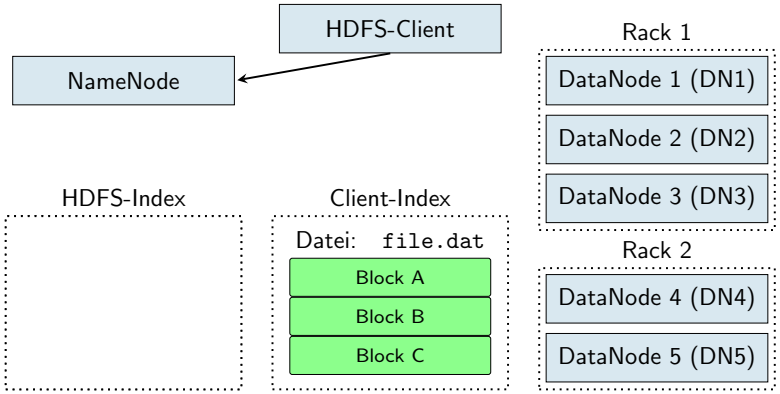
- System-Konfiguration
  - 1x HDFS-Client
  - 1x NameNode
  - 5x DataNodes (Rack 1: DN1–3, Rack 2: DN4–5)

Hadoop Distributed File System (HDFS) — Schreiben



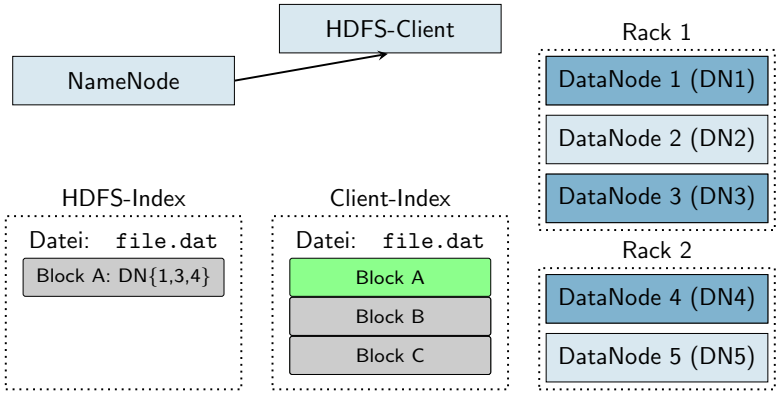
- HDFS-Client legt die aus drei Blöcken (Block A, B und C) bestehende Datei `file.dat` im HDFS an

Hadoop Distributed File System (HDFS) — Schreiben



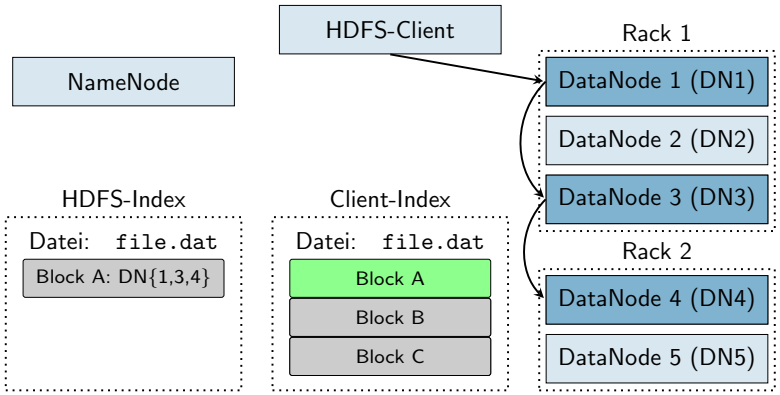
1. HDFS-Client → NameNode:
   Anforderung eines sog. Lease (dt. *Miete*) für das Schreiben der Datei `file.dat`

Hadoop Distributed File System (HDFS) — Schreiben



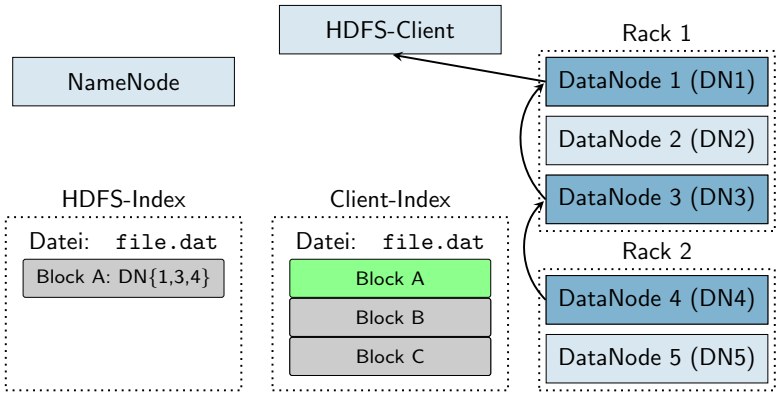
2. NameNode → HDFS-Client:
   Erteilung des Lease, Erzeugung einer Block-ID für den ersten Block (Block A), Zuteilung der Replikate (DN1, DN3 und DN4)

Hadoop Distributed File System (HDFS) — Schreiben



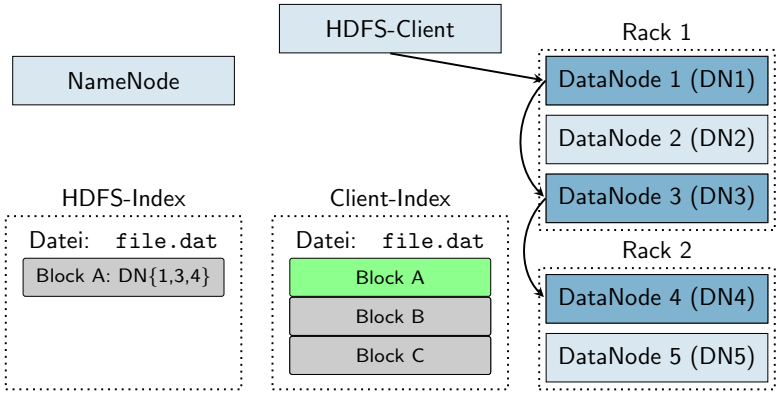
3. „Daten-Pipeline“ zur Vorbereitung der Schreiboperation*en* von Block A:
   HDFS-Client — DN1 — DN3 — DN4

Hadoop Distributed File System (HDFS) — Schreiben



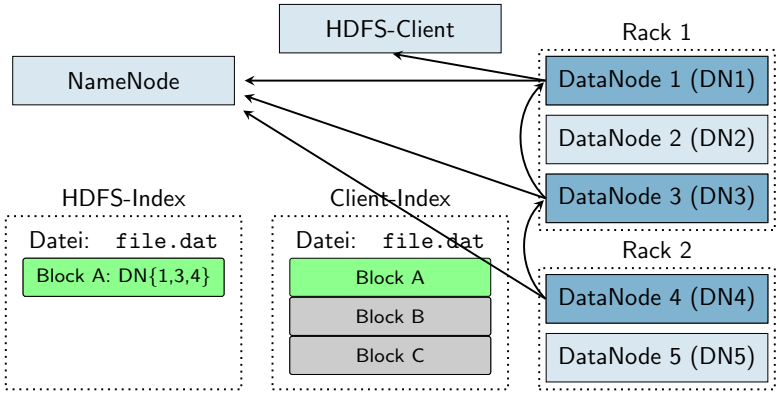
3. „Daten-Pipeline“ zur Vorbereitung der Schreiboperation*en* von Block A:
   HDFS-Client — DN1 — DN3 — DN4

Hadoop Distributed File System (HDFS) — Schreiben



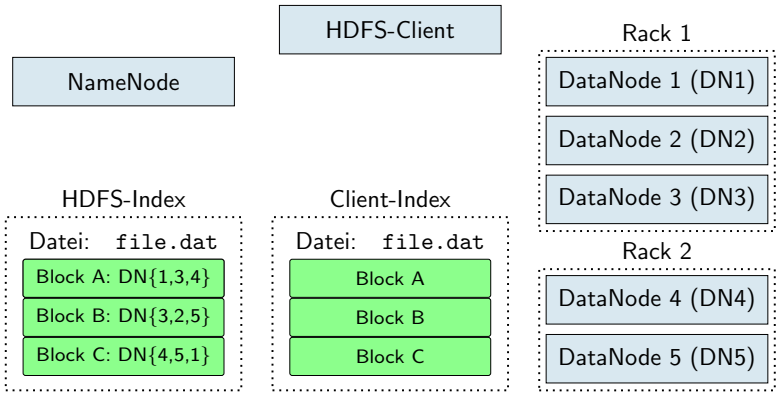
4. Durchführung der Schreiboperationen:
HDFS-Client sendet Block A an DN1
DN1 sendet empfangenen Block A an DN3
DN3 sendet empfangenen Block A an DN4

Hadoop Distributed File System (HDFS) — Schreiben



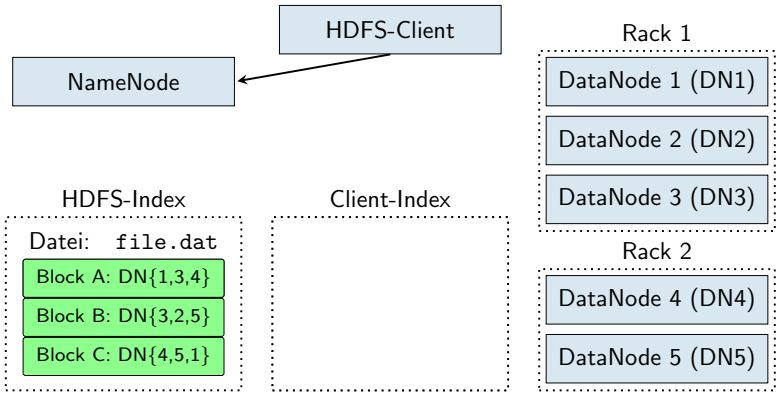
5. Bestätigung der Schreiboperationen:
Jede DataNode bestätigt das erfolgreiche Schreiben von Block A gegenüber dem NameNode *und* entlang der Pipeline (Abbau)

Hadoop Distributed File System (HDFS) — Schreiben



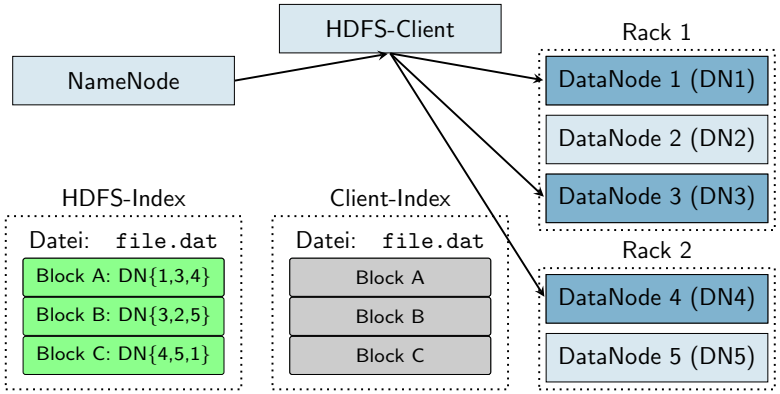
- HDFS-Client → DataNodes:
Analog werden die restlichen Blöcke der Datei vom HDFS-Client an die durch den NameNode zugeordneten DataNodes verschickt

Hadoop Distributed File System (HDFS) — Lesen



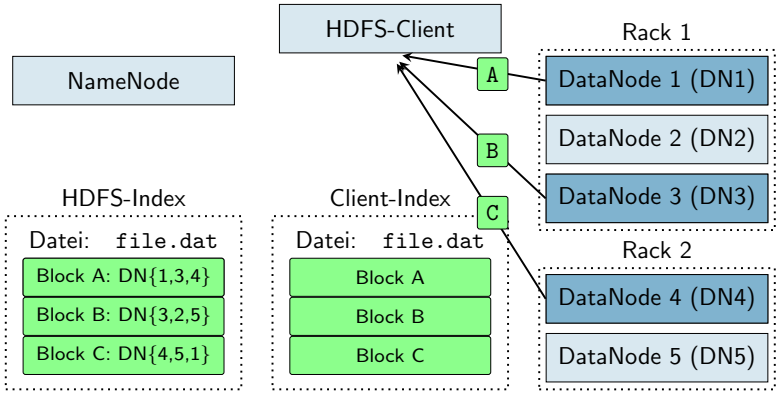
1. HDFS-Client → NameNode:
Anforderung der DataNodes-Liste: Alle DataNodes, die Blöcke der zu lesenden Datei `file.dat` speichern

Hadoop Distributed File System (HDFS) — Lesen



2. NameNode → HDFS-Client, HDFS-Client → DataNodes:
Client erhält DataNodes-Liste und wählt den ersten DataNode für jeden der Datenblöcke

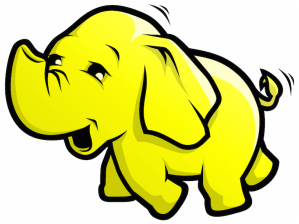
Hadoop Distributed File System (HDFS) — Lesen



3. DataNodes → HDFS-Client:
HDFS-Client liest die Blöcke sequentiell, DataNodes senden die angeforderten Blöcke an den HDFS-Client

Hadoop Distributed File System (HDFS)

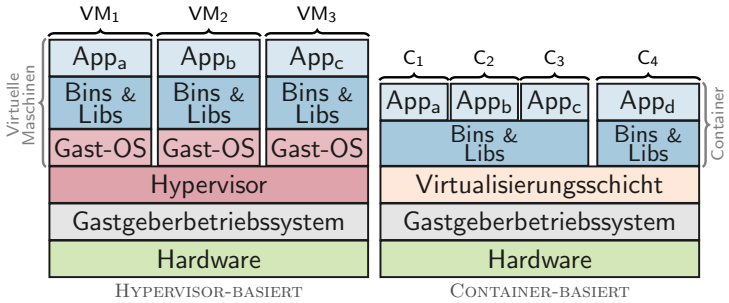
- (Weitere) HDFS-Details
  - Herzschlag-Nachrichten (engl. heartbeat): DataNodes → NameNode
    - → Alle drei Sekunden (Default) ein Herzschlag
    - → Grundlast bei sehr großen Clustern
  - Block-Report: NameNode generiert Metadaten aus den Block-Reports
    - → Replikationsfaktor sicherstellen
    - → Löschen ungenutzter Blöcke
  - NameNode
    - → *Die Sollbruchstelle des Systems?*



- Informationen und Links
  - Apache Hadoop: HDFS Architecture
  - Shvachko et al.: The Hadoop distributed file system

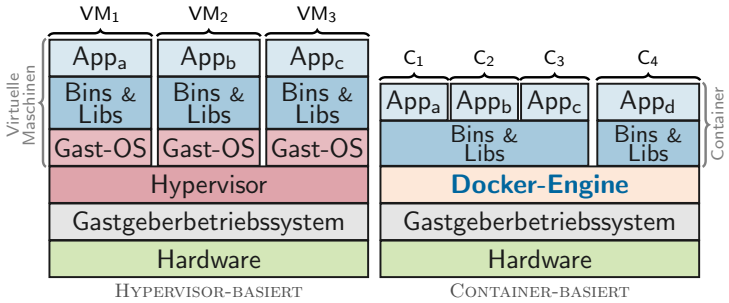
Überblick

Virtualisierungsformen im Vergleich

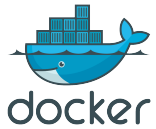


- Hypervisor-basierte Virtualisierung (Vollvirtualisierung)
  - Stärken liegen in der Isolation unabhängiger virtueller Maschinen
  - Erlaubt Virtualisierung von kompletten Betriebssystemen
- Container-basierte Virtualisierung
  - Leichtgewichtig: Hypervisor entfällt, kleinere Abbilder
  - Benötigt unter Umständen angepassten Betriebssystem-Kernel

Virtualisierungsformen im Vergleich



- Container-Betriebssystemvirtualisierungsstellvertreter
  - {Free,Open,Net}BSD: FreeBSD jail, sysjail
  - Windows: Turbo, Sandboxie
  - Linux: OpenVZ, LXC
- Im Rahmen dieser Übung betrachtet: **Docker**



Docker

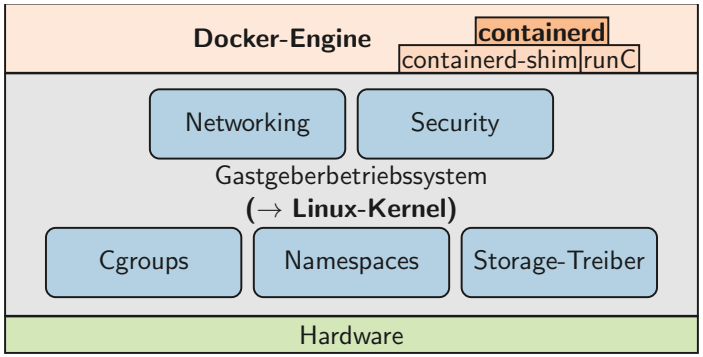
Video: **„What is Docker?“**

Kurzvortrag von Docker-Erfinder Solomon Hykes

(Kopie: `/proj/i4mw/pub/aufgabe3/What_is_Docker.mp4`, Dauer: 7:15 Min.)

Docker-Architektur

Überblick



- Docker setzt auf bereits existierenden Linux-Komponenten auf
- Dominierende Komponenten
  - Ressourcenverwaltung: Control Groups
  - Namensräume
  - Überlagerte Dateisysteme

  } **containerd & runC**

Control Groups

- 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

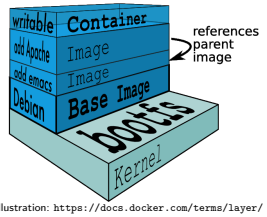
Tejun Heo
Control Group v2
`https://www.kernel.org/doc/Documentation/cgroup-v2.txt`, 2015.

Namensräume

- Namensräume werden zur Isolation von Anwendungen auf unterschiedlichen Ebenen herangezogen
- **Dateisysteme**
  - Jedes Dateisystem benötigt eigenen Einhängepunkt, 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

Dockerizing: Anwendung → Container

- Unterscheidung
  - Docker-Abbild: Software-Basis zum Instanziieren von Docker-Containern
  - Docker-Container: Instanziiertes Docker-Abbild in Ausführung
- Inhalt eines Docker-Containers
  - Dateisystem
  - Systembibliotheken
  - Shell(s)
  - Binärdateien



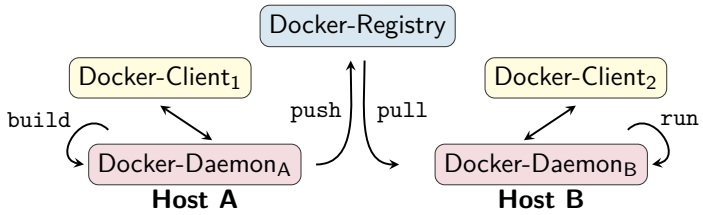
Quelle der Illustration: `https://docs.docker.com/terms/layer/`

- **Dockerizing:** „Verfrachten“ einer Anwendung in einen Container
  - Instanziieren 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

Docker-Arbeitsablauf

- Git-orientierter Arbeitsablauf
  - Ähnliche Befehlsstruktur (z. B. `pull`, `commit`, `push`)
  - Git Hub ⇔ Docker Hub
- Typischer Arbeitsablauf

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



Docker-Registries und -Repositorys

- Von Docker, Inc. bereitgestellte Registry: **Docker Hub**
  - Cloud-Service zur Verwaltung von Docker-Abbildern bzw. -Anwendungen
  - Registrieren bzw. Anlegen eines Benutzerkontos notwendig
  - Anzahl kostenloser, **öffentlicher** Repositorys nicht begrenzt
  - Nur ein privates Repository kostenlos
- **Private Registry** (hier: I4-Docker-Registry)
  - Ermöglicht das Verwalten garantiert nicht-öffentlicher Repositorys
  - 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 ausfuehren, siehe naechste 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`)

Umgang mit der Registry

- Abbild aus Repository herunterladen und direkt verwenden/verändern
  1) Vorgefertigtes Abbild aus Repository auschecken

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

  Hinweis: `TAG` nur optional, wenn Image mit Default-Tag (= `latest`) existiert.
  2) Container starten (mehr ab Folie 5–22); darin evtl. Änderungen vornehmen

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

  Mit `/bin/bash` als `COMMAND` können im Container über die Shell beliebige Programme via Paket-Manager installiert werden, z. B. `apt-get -yq install vim`.
- Optionale Schritte (nur falls Änderungen erfolgt sind, die erhalten bleiben sollen)
  3) Änderungen persistent machen und Abbild (lokal!) erzeugen

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

  4) 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`

Docker-Abbilder bauen

Skriptbasiert (1)

- In der Praxis: **Dockerfiles**
  - Rezepte zum skriptbasierten Bauen eines Abbilds
  - Zeilenweises Abarbeiten der darin befindlichen Instruktionen
- Vordefinierte, voneinander unabhängige **Docker-Instruktionen**
  - `FROM <IMAGE>[:<TAG>]` ↦ Basisabbild auswählen (obligatorisch)
  - `EXPOSE <PORT> [<PORT>...]` ↦ Container-übergreifende Port-Freigabe
  - `RUN <COMMAND>` ↦ Ausführen eines Befehls (in *Shell-Form*)
  - `ENTRYPOINT [<EXE>, <PARAM-1>, ...]` ↦ 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>, ...]`
  - `COPY <SRCs> <DST>` ↦ Dateien/Verz. ins Container-Dateisystem kopieren
  - `...` [↪ vollständige Referenz: `https://docs.docker.com/reference/builder/`]

Docker-Abbilder bauen

Skriptbasiert (2)

- 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; 2) Port 18084 freigeben
  3) Benutzer `mwcc` erstellen, diesen zur Gruppe `users` hinzufügen und Shell setzen
  4) Basisverzeichnis setzen (`/opt/mwcc` und `lib`-Unterverzeichnis existieren bereits)
  5) Log-Verzeichnis erstellen, Benutzerrechte setzen und 6) JAR-Datei hineinkopieren
  7) Ausführenden Benutzer, 8) Einstiegspunkt und 9) Standardargumente setzen

Docker-Abbilder

- 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

Zustandsänderungen

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

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

→ Für `IMAGE` kann `NAME[:TAG]` (vgl. Folie 5–18) oder die Image-ID eingesetzt werden.

- 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...>`
  - Attachen ↦ `docker attach --sig-proxy=false <CONTAINER-IDs...>`
    Hinweis: `-sig-proxy=false` nötig, um mit `Ctrl-c` detachen zu können

Docker-Container

Zustandsanalysen

- Möglichkeiten der Container-Analyse
  - Logs (≙ 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`

  → **Sämtliche** Events am Docker-Server anzeigen (in Echtzeit):

```
$ docker events
```

Docker-Container

- Nachträglich Befehle ausführen (z. B. zu Debugging-Zwecken)
  - Weiteren Befehl innerhalb eines bereits laufenden Containers starten

```
$ docker exec -d <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 ...
```

Docker-Volumes

`docker volume`

- Daten innerhalb eines Containers sind an dessen Lebensdauer gebunden
- Daten, die über die Container-Lebensdauer hinweg erhalten bleiben sollen, können beispielsweise in ein Docker-Volume gelegt werden
- Befehlsübersicht
  - Volume erstellen

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

  - Volumes auflisten

```
$ docker volume ls
```

  - Volume löschen

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

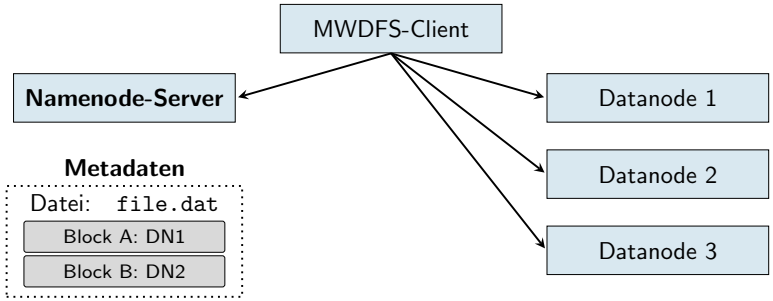
  - Neuen Container mit einem existierenden Volume starten

```
$ docker run -d --mount source=<VOLUME-NAME>,target=<MOUNT-POINT> \
<IMAGE-ID> [<COMMAND> (sonst: '/bin/bash')]
```

Hinweis: Beim Einhängepunkt (`<MOUNT-POINT>`) ist darauf zu achten, dass der Benutzer im Container Schreibrechte auf das korrespondierende Verzeichnis hat. Die Schreibrechte werden nur in ein neues Volume übernommen.

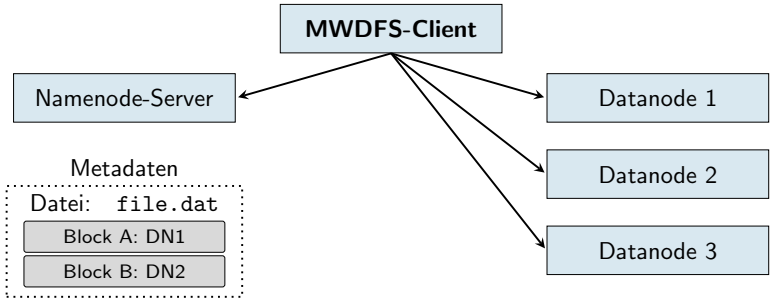
Überblick

Übersicht



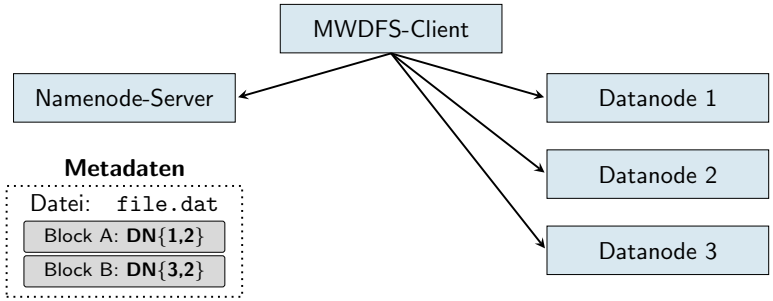
- **Namenode-Server**
  - **Metadaten**
  - Datei-Operationen (Anlegen, Anzeigen, Löschen)
  - Leases für Schreibzugriffe

Übersicht



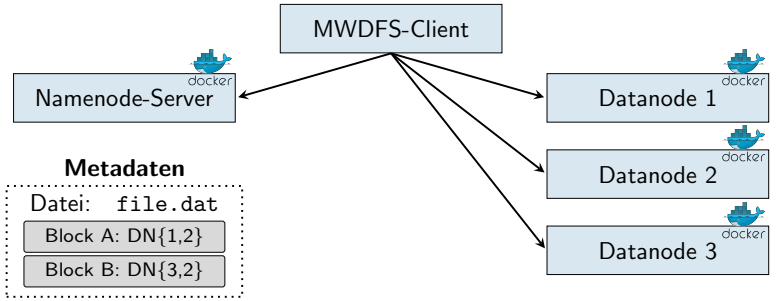
- **MWDFS-Client**
  - Datenzugriff
  - Datei-Operationen (Anlegen, Anzeigen, Löschen)

Übersicht



- **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}

Übersicht



- **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

{S,Des}erialisierung in 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
  - Erzwingen des Schreibvorgangs auf Datenträger (HDD/SSD) mittels Aufruf von `force()` an `FileChannel`-Objekt
    → `boolean`-Parameter von `force`: 'true' := Dateiinhalt **und** -metadaten schreiben

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

```
long readLong = dis.readLong();
dis.close();
```

JAX-RS: Übertragung von Binärdaten

- Datanodes empfangen (`POST`) und senden (`GET`) Blockdaten als Binärdaten
  - Server-seitiger Datanode-Code: `mw.datanode.MWDataNodeWebService` → Pub-Verzeichnis
- 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;
}
```

Docker-Hilfsskripte

- **Hilfsskripte** liegen in OpenStack-VM bereit unter `/usr/local/bin`
- Verfügbare Skripte
  - Löschen aller {gestoppten,ungetaggten} Docker-Container

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

  - Alle Container stoppen und Docker-Daemon neustarten

```
$ docker-full-reset
```

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

```
$ docker-images-push
```

  - I4-Docker-Registry durchsuchen

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