ScaDS DRESDEN LEIPZIG

UNIVERSITÄT LEIPZIG

AUS TRADITION
GRENZEN ÜBERSCHREITEN

Cloud Data Management

Kapitel 3: Verteilte Dateisysteme (Teil 1)

Dr. Eric Peukert

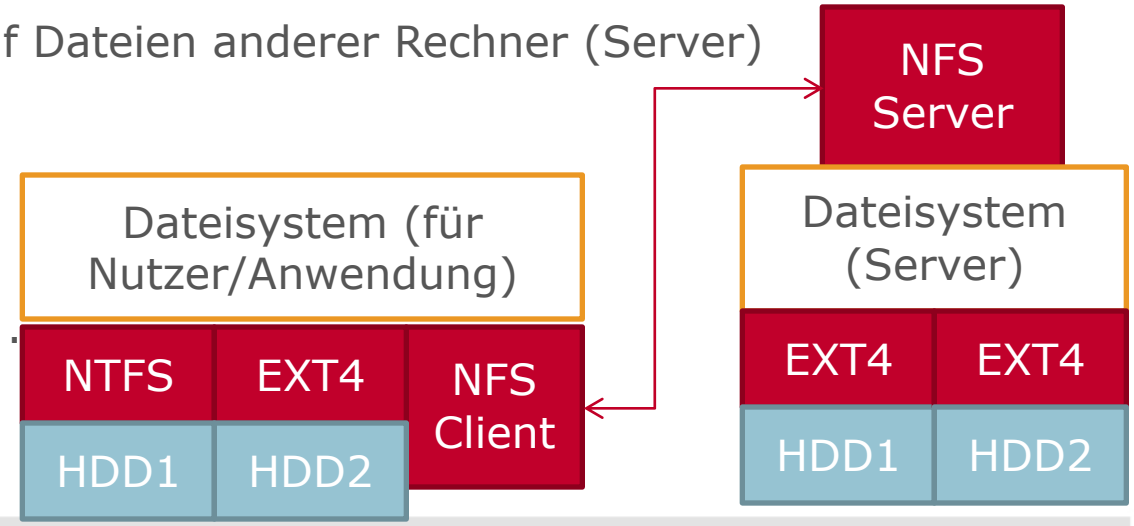
Wintersemester 2017

Universität Leipzig, Institut für Informatik
http://dbs.uni-leipzig.de

Inhaltsverzeichnis

Dateisysteme und verteilte Dateisysteme

- Dateisystem
  - System für permanente Datenspeicherung
  - Zugriffsschicht für physischen Datenträger (HDD, DVD, ...)
  - Basisobjekt: Datei
  - eindeutig referenziert durch Namen und (hierarchischen) Pfad
  - Beispiele: FAT32, NTFS, ext4, ...
- Verteiltes Dateisystem
  - ermöglicht Zugriff auf Dateien anderer Rechner (Server)
  - Problemfälle
    - Concurrency
    - Replication
    - Caching
  - Beispiele: DFS, NFS, ...



Notwendigkeit neuer Dateisysteme für Cloud

	Lokal / Netzwerk	Cloud
Nutzer	Anwender, (lokale) Programme	Verteilte Large-Scale-Berechnungsprogramme
Beispiele	Textverarbeitung Fotoverwaltung	Suchmaschine (Index, PageRank, ...), Soziales Netzwerk (Clustering, FoaF, ...)
Anzahl der Dateien	sehr viele (>>1,000,000)	"wenige" (≈ 1,000,000)
Größe der Dateien	kleine (KB-MB)	große (GB, TB, …)
Lesezugriff	teilw. concurrent, komplett	concurrent, streaming
Schreibzugriff	mehrfach Überschreiben	write-once, Anhängen von Daten (append), concurrent writes

Annahmen und Ziele

- Hardware Ausfall
  - Ausfall ist die Norm (nicht die Ausnahme)
  - Gewährleistung von Datensicherheit

Hersteller	Typ	Typ	Kapazität	Anzahl	"Drive Days"	Ausfälle	AFR
Seagate	ST4000DM000	Desktop HDD.15	4 TByte	34.729	2.849.179	198	2,54%
HGST	HMS5C4040ALE640	MegaScale 4000	4 TByte	7.075	637.116	10	0,57%
HGST	HDS5C3030ALA630	Deskstar 5K3000	3 TByte	4.552	410.112	6	0,53%
HGST	HDS722020ALA330	Deskstar 7K2000	2 TByte	4.264	399.203	19	1,74%
HGST	HMS5C4040BLE640	MegaScale DC 4000.B	4 TByte	3.091	278.190	0	0,00%
HGST	HDS5C4040ALE630	Deskstar 5K4000	4 TByte	2.706	243.312	7	1,05%
Seagate	ST6000DX000	Barracuda?	6 TByte	1.882	169.380	0	0,00%
WD	WD30EFRX	WD Red	3 TByte	1.054	94.384	8	3,09%
HGST	HDS723030ALA640	Deskstar 7K3000	3 TByte	998	89.923	2	0,81%
WD	WD60EFRX	WD Red	6 TByte	458	41.220	6	5,31%
Seagate	ST4000DX000	Barracuda?	4 TByte	207	18.945	5	9,63%
Toshiba	MD04ABA400V	Surveillance	4 TByte	146	13.108	0	0,00%
WD	WD20EFRX	WD Red	2 TByte	133	11.617	4	12,57%
Toshiba	DT01ACA300	Desktop	3 TByte	47	4.230	1	8,63%
WD	WD40EFRX	WD Red	4 TByte	46	4.140	0	0,00%
HGST	HUH728080ALE600	Ultrastar He8	8 TByte	45	4.050	0	0,00%
Seagate	ST31500541AS	Barracuda LP 5900	1,5 TByte	45	5.961	0	0,00%
Toshiba	MD04ABA500V	Surveillance	5 TByte	45	4.050	0	0,00%

https://www.recode.net/2017/11/1/16592692/fog-computing-cloud-centralized-infrastructure-software

Annahmen und Ziele (2)

- Streaming Data Access
  - Verteiltes, sequentielles Lesen (blockweise)
  - Batch Processing statt interaktiver Nutzung
- Sehr große Datensätze
  - Millionen von Dateien mit je GB-TB
  - Hohe Gesamtbandbreite nötig
- Einfaches Kohärenzmodell
  - „Write-once-read-many“ Zugriffe
  - Vereinfachung von Daten-Kohärenz-Problemen für hohen Datendurchsatz

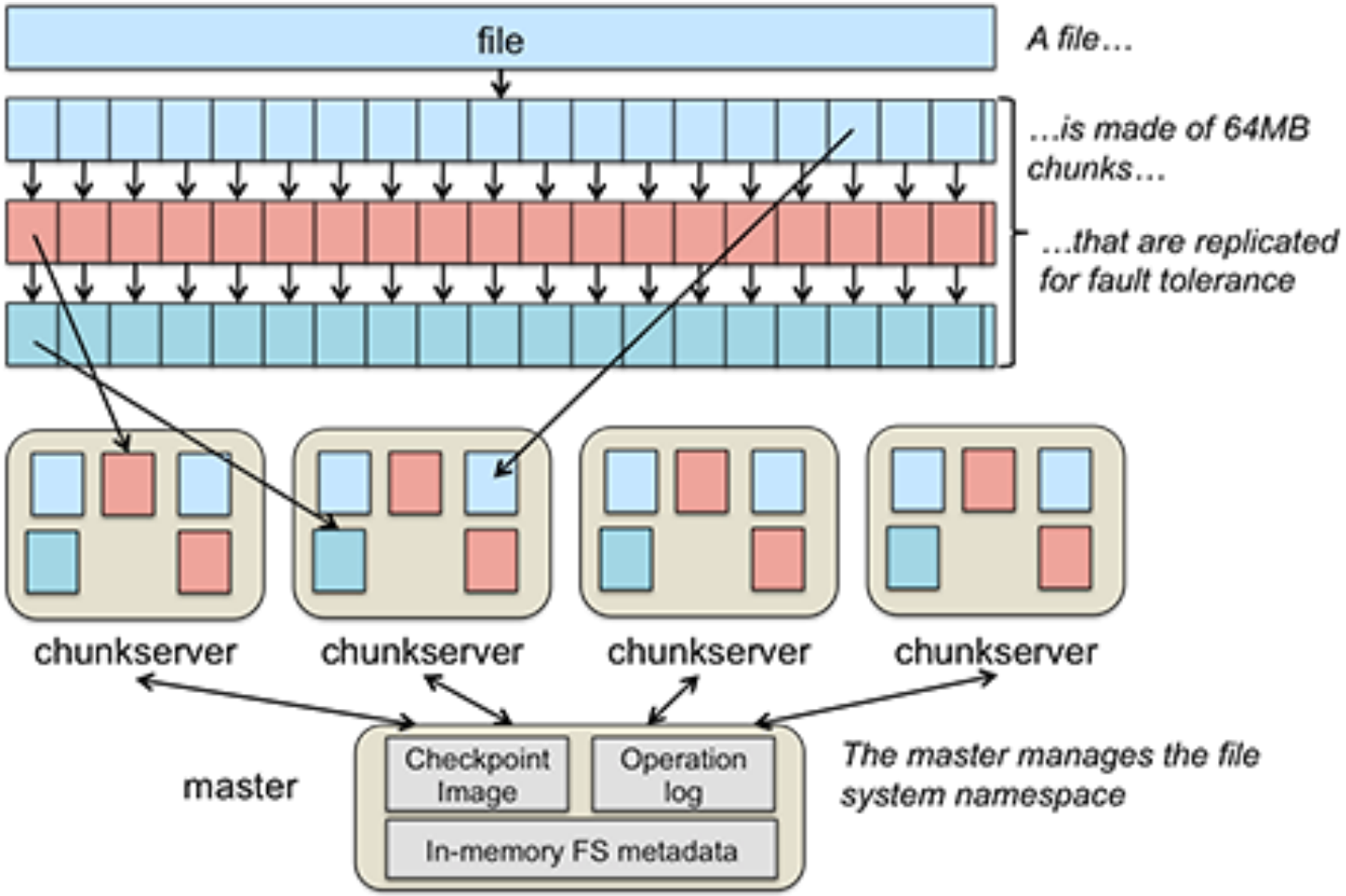
Annahmen und Ziele (3)

- Datennahe Berechnung
  - Berechnung soll „nah“ an den zu verarbeitenden Daten stattfinden
  - Wissen über Speicherort der Daten notwendig (Lokalität)
- Portabilität
  - Unterstützung heterogener Hard- und Software

Google File System

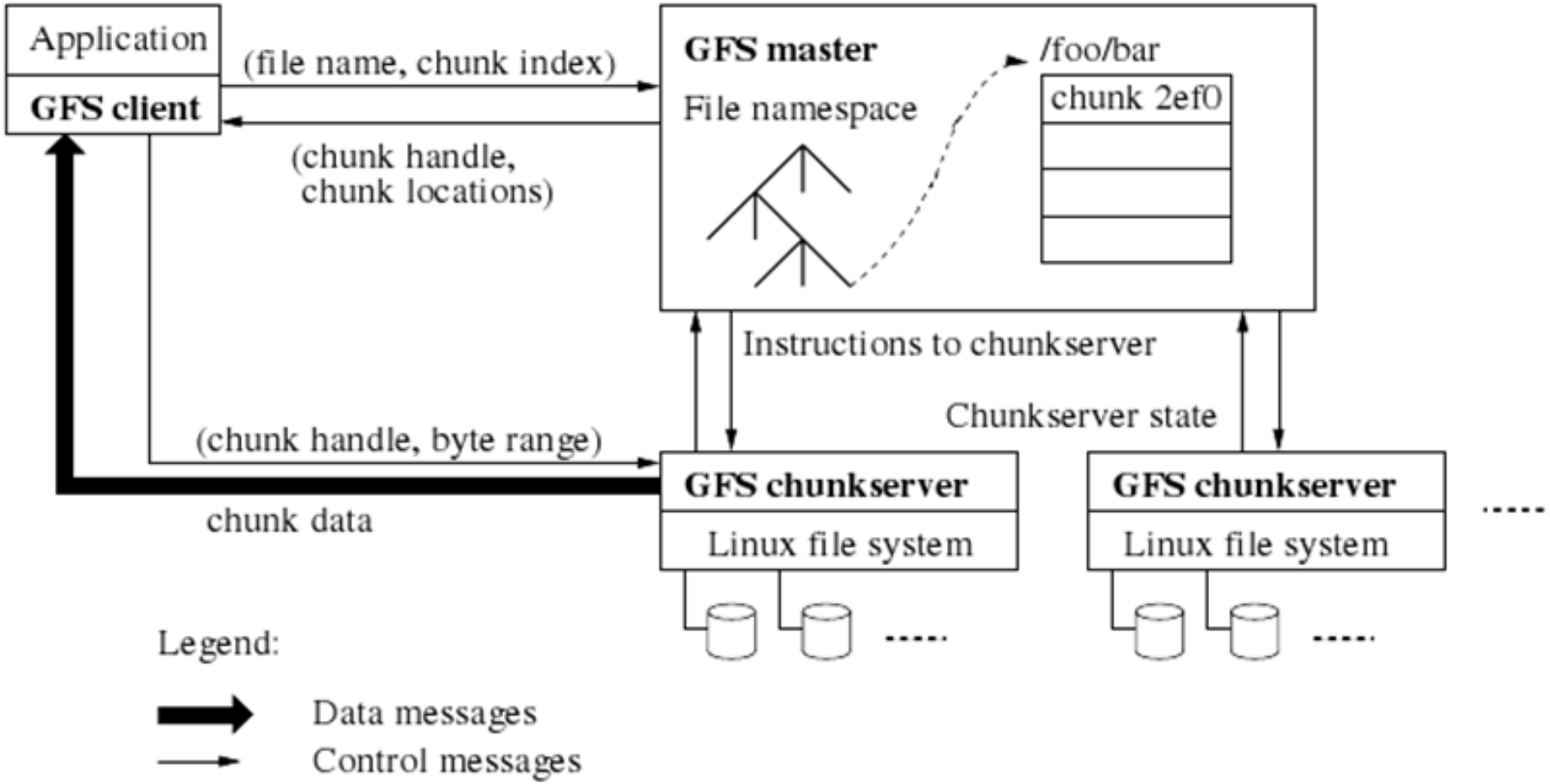
- Proprietäres, verteiltes Linux-basiertes Dateisystem
  - Hochskalierend: tausende Festplatten, mehrere 100TB
  - Open Source-Alternative: Hadoop Distributed File System
- Netzknoten: „billige" Standardhardware (kein RAID)
- Optimiert für Streaming Access (Write-once-read-many)
- Physische Datenpartitionierung in Chunks (Default: 64 MB)
  - Verteilung über mehrere Chunkserver (und Replizierung jedes Chunks)
- Master-Server
  - Mapping: Datei → Chunk → Node
  - Replikat-Management: Default 3 Chunk-Kopien (in 2 Racks)
  - Anfragebearbeitung, Zugriffsverwaltung

GFS Data Chunks



https://www.cs.rutgers.edu/~pxk/417/notes/16-dfs.html

Google File System: Architektur

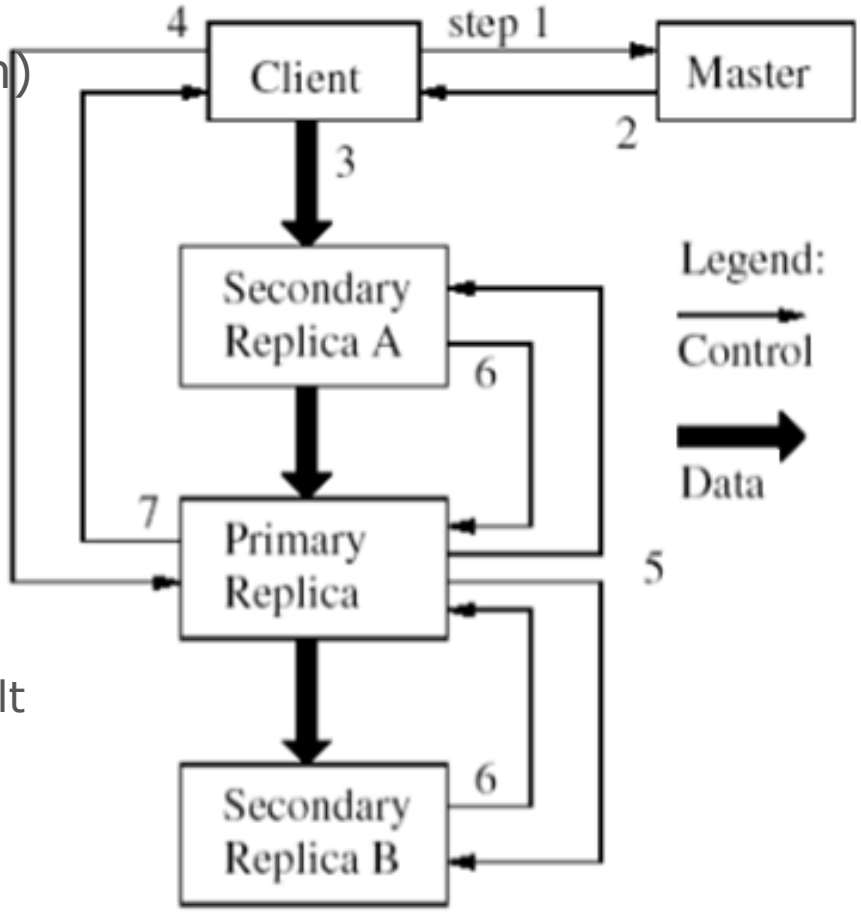


Master

- Metadata-Management
  - Mapping: Datei → Chunk → ChunkServer (Node)
  - Alles im Hauptspeicher (performant)
    - 64 byte pro Chunk, "wenige" Dateien
- Logfile
  - persistentes Logging kritischer Metadaten-Updates
  - Repliziert, Checkpoints (Recovery)
- Single-Master-Problem (Single Point of Failure = SPOF, Bottleneck)
  - Shadow Masters
    - haben ("recent") Kopie des Mappings; springen bei Ausfall ein
  - Reduzierter Workload für Master
    - nur Metadaten
    - große Chunksize (64MB), damit wenige Metadaten / Interaktion / Netzwerk Overhead
    - Authority-Weiterreichung an Primary Replicas bei Datenänderung (Lease)

Datenmanipulation

- Mutation (Schreiben oder Anhängen)
  - muss für alle Replikate durchgeführt werden
- Lease Mechanismus
  - Master bestimmt ein Replica zur Koordinierung (“lease for mutation”)
  - Primary Replica sendet Folge von Mutations-Operationen an alle Secondary Replicas
  - Reduzierter Master-Workload
  - Datenfluss von Kontrollfluss entkoppelt



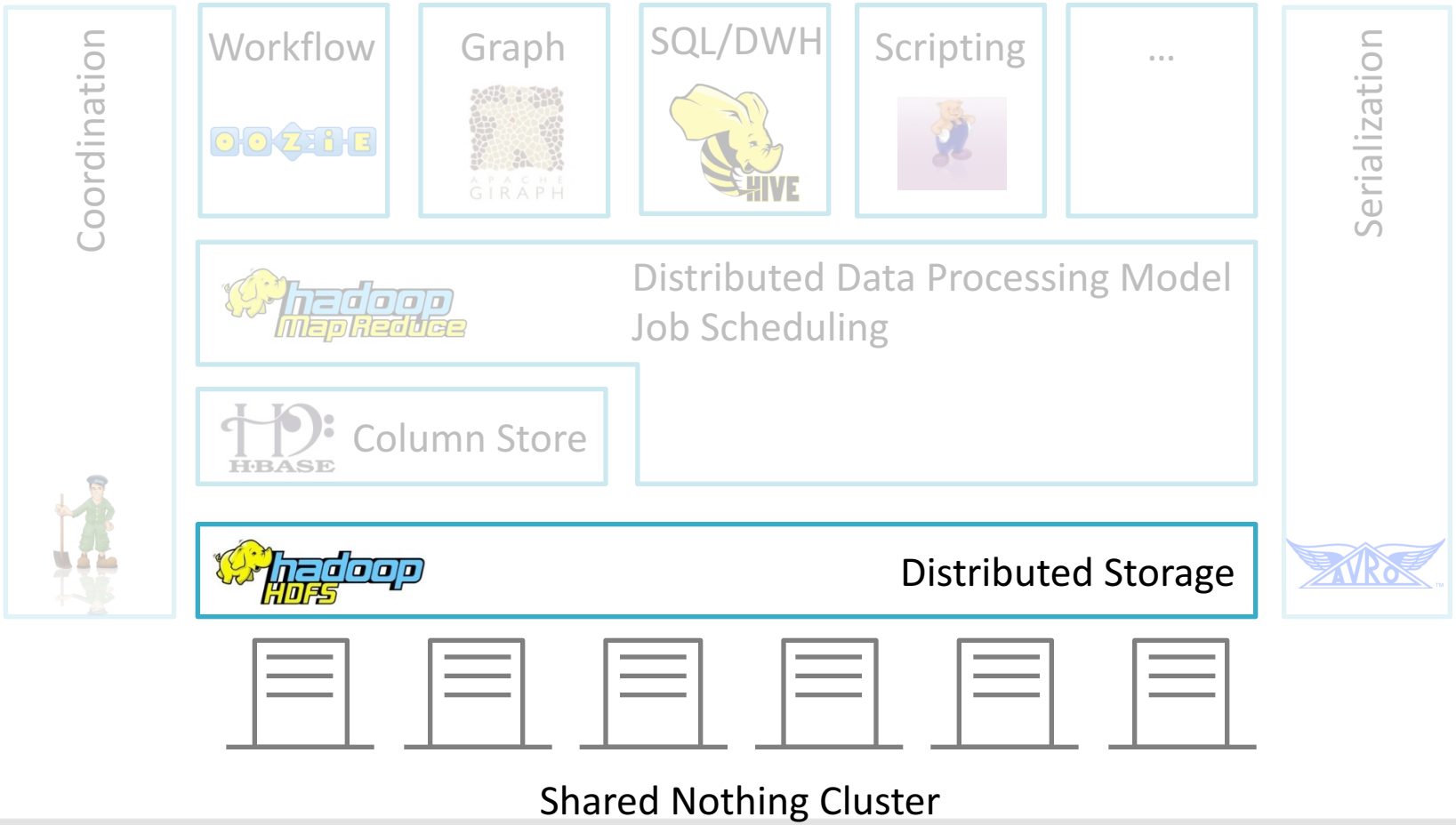
[GGL03]

Datenmanipulation(2)

- (atomic) Record Append-Operation
  - GFS hängt Daten von Client an File an
  - Schreiben koordiniert durch Primary
  - GFS bestimmt Offset, damit parallele Schreibzugriffe möglich
  - optimiert für Multiple-Producer-Single-Reader-Queues (z.B. MapReduce)
  - Im Fehlerfall bei einem der Replica
    - Schreiben über Primary wird wiederholt
    - Replicate können unterschiedlich sein und Duplikate einer Schreiboperation enthalten (at least once)

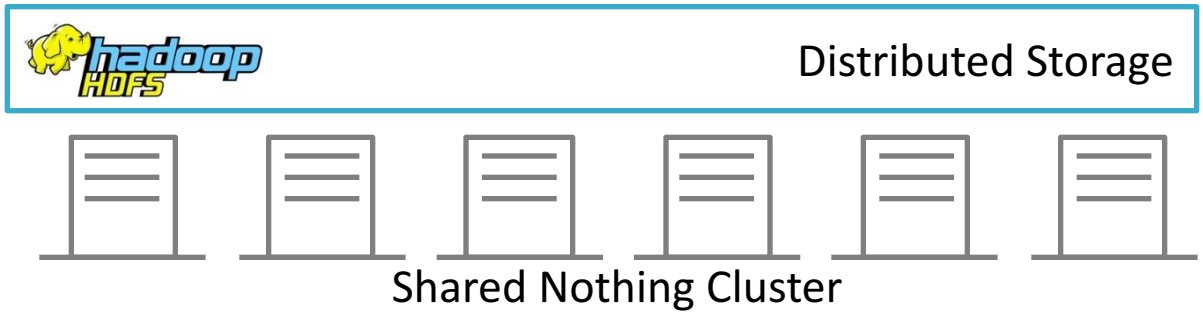
Hadoop Distributed Filesystem

Apache Hadoop Ecosystem

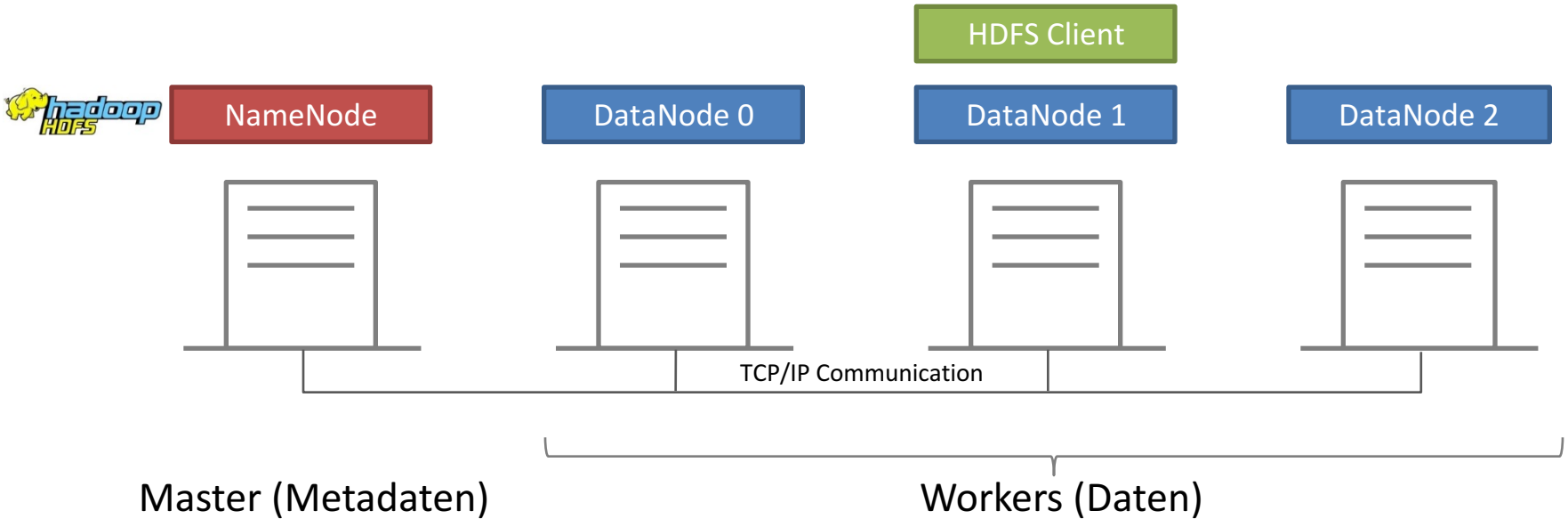


Apache HDFS

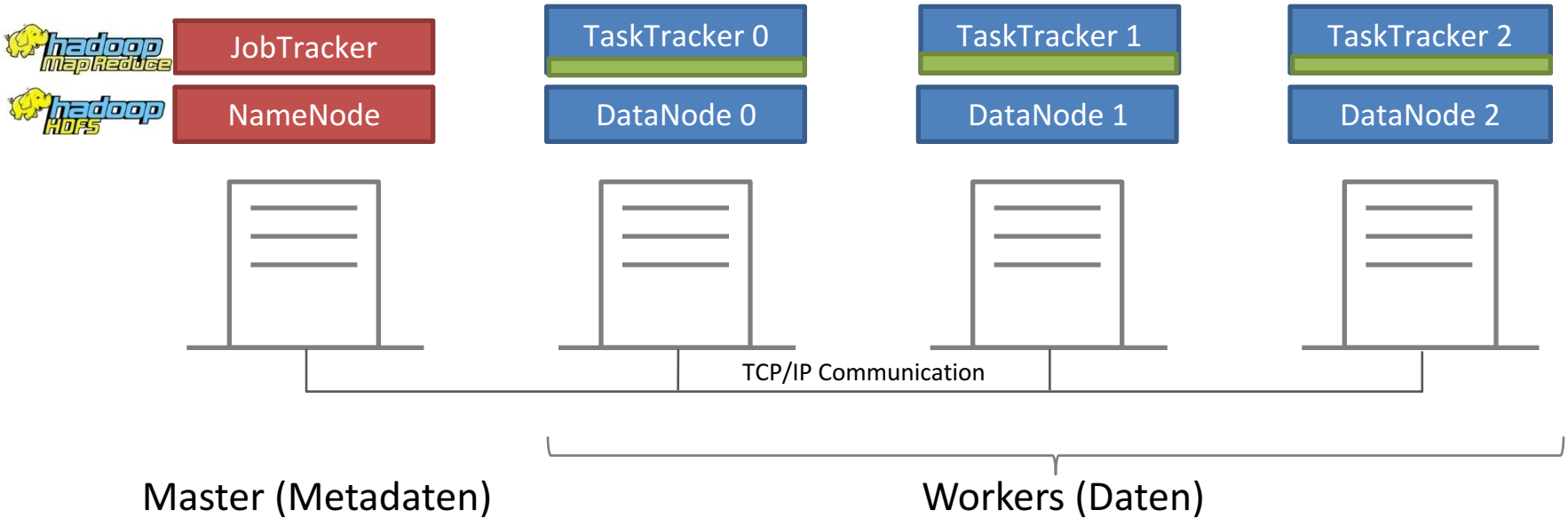
- Verteiltes Dateisystem von Hadoop
- Modul des Apache Hadoop Core Top-Level Project
- Open Source unter Apache Lizenz, Version 2.0
- Initiale Entwicklung durch Yahoo!
- Vollständig in Java implementiert
- Non-POSIX konform für verbesserte Performanz
- Läuft typischerweise auf GNU/Linux Betriebssystemen



Architektur – Überblick



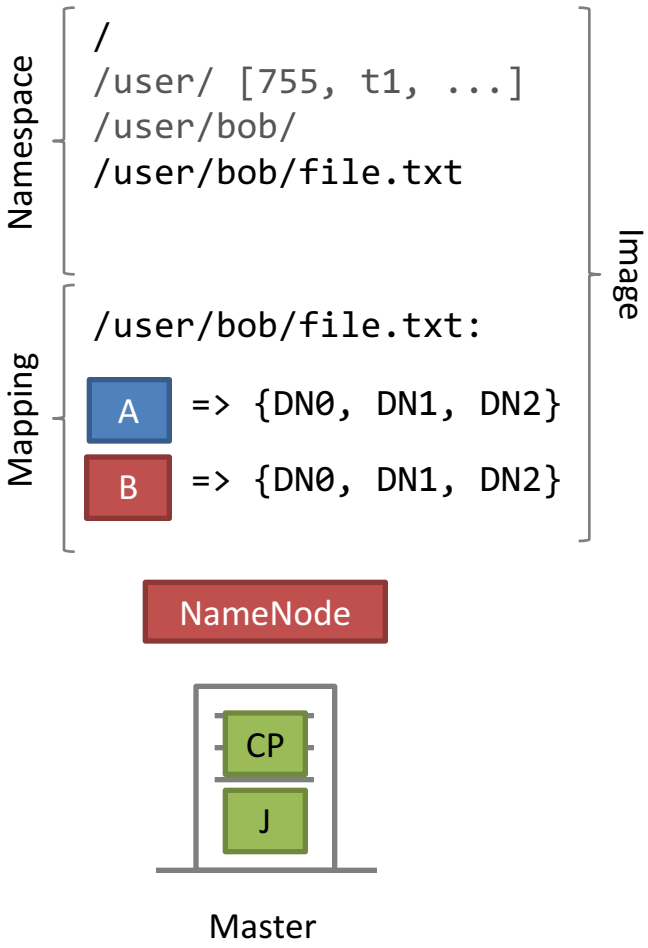
Architektur – Überblick



Architektur – Blöcke

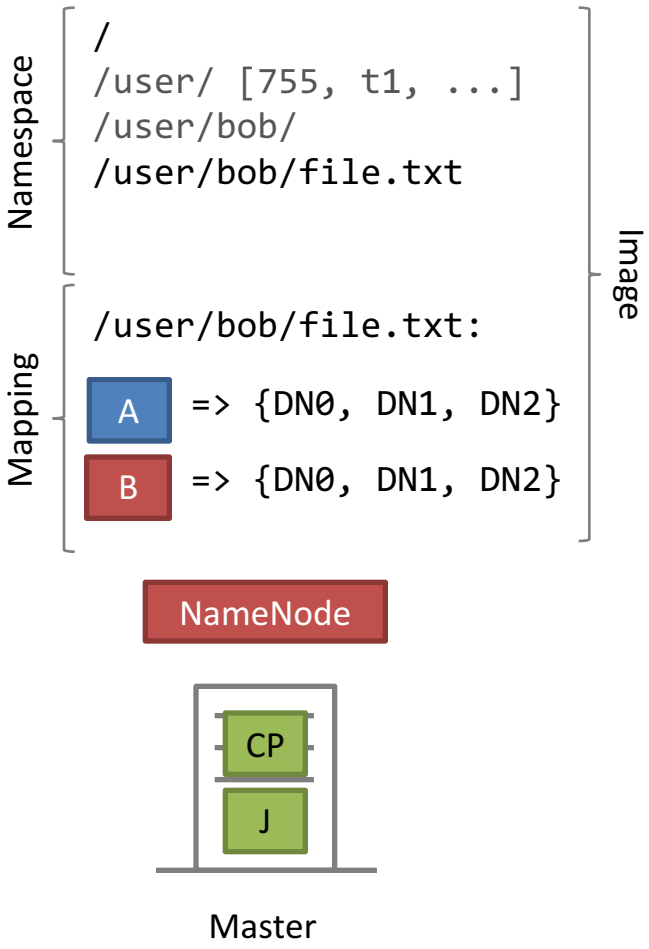
- Block: Minimale Datenmenge, die eine Festplatte/Dateisystem lesen o. schreiben kann
  - 512 Byte für Festplatten
  - 4 Kilobyte für lokale Dateisysteme (z.B. ext4)
  - 64 Megabyte für HDFS (default, 128MB typisch)
- Warum 64 MB?
  - Minimierung der Kosten für disk seeks
  - Übertragung großer Dateien möglichst mit „disk speed“
- HDFS Dateien werden in Blöcke / Chunks zerlegt
  - Blöcke/Chunks/Splits sind die Basiseinheit für Replikation (kein RAID)
  - Split: Eingabe für Map- oder Reduce Task
  - Dateigröße nicht durch lokales Dateisystem limitiert
  - „Write-once-read-many access“ Modell
  - `/user/bob/file.txt [128MB] ->` 

Architektur – NameNode (NN)



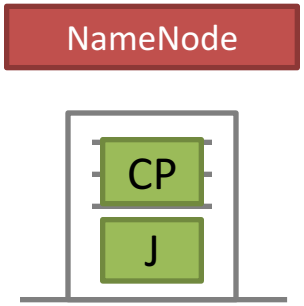
- Verwaltet Namespace-Tree des Dateisystems
  - Eindeutige **namespaceID**
  - Ordner und Dateien repräsentiert durch **inodes**
  - inode: Datei- und Verzeichnisrechte (permissions), Zugriffs-& Modifizierungszeit, space quotas
- Verwaltet Mapping der Blöcke zu DataNodes (DN)
- Namespace + Mapping = **Image (fsimage)**
  - Image wird vollständig im RAM des NN gehalten

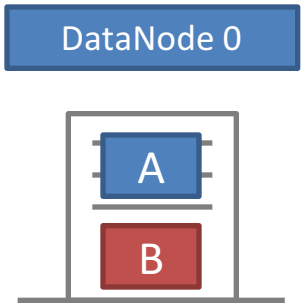
Architektur – NameNode (NN) (2)



- Persistenter Namespace = **Checkpoint (CP)**
- Modification-Log des Image = **Journal (J) / Edit Log**
  - Write-Ahead-Log enthält Dateisystem Änderungen
- Während (Neu)start des NN wird das Image aus dem Checkpoint und Journal wiederhergestellt
- Checkpoint + Journal sind kritische Komponenten
  - Typischerweise an verschiedenen (remote) locations gespeichert

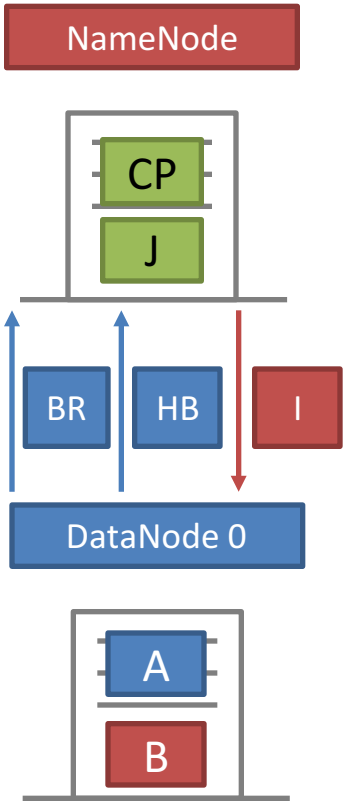
Architektur – DataNode (DN) (1)





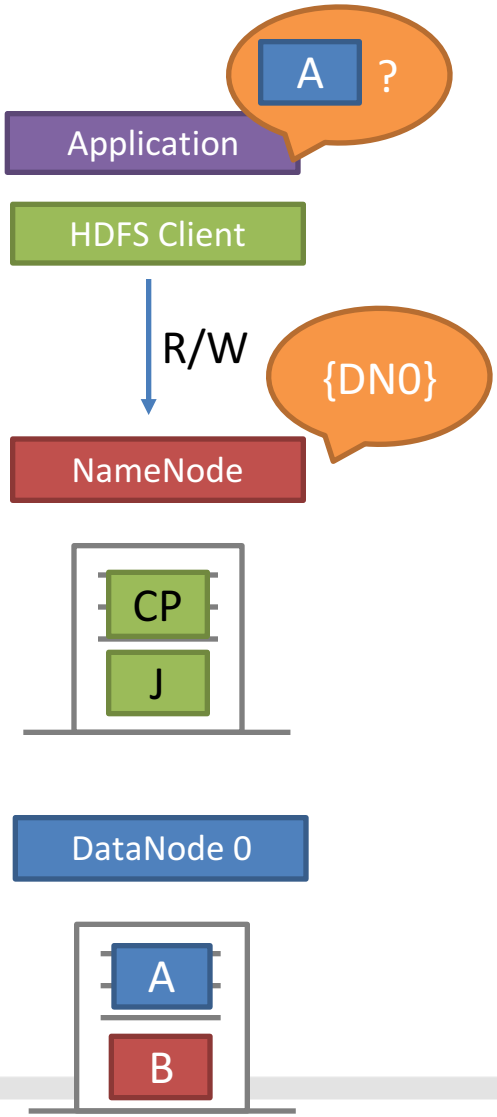
- Identifiziert durch eindeutige **storageID**
  - Zuweisung bei erster Verbindung zu NN
  - Unabhängig von IP Adresse oder Port
- Registriert sich beim NN (**storage ID** + **namespaceID**)
- Speichert Block Replicas im lokalen Dateisystem
- Block wird durch zwei Dateien repräsentiert
  - Eigentliche Daten
  - Block Metadaten + Checksumme

Architektur – DataNode (DN) (2)



- Sendet stündliche **block reports (BR)** zum NN für Mapping-Updates
  - Für jedes Block Replica: BlockID, Länge, …
- Sendet **heartbeats (HB)** zum NN (3 Sekunden)
  - Timeout nach 10 min, DN wird als „out of service" betrachtet, NN setzt Blöcke auf 'nicht verfügbar
  - Weitere Informationen: Gesamtspeicherkapazität, genutzter Speicheranteil, Anzahl aktueller Datentransfers
- Erhält **instructions (I)** vom NN durch „Heartbeat Replies"
  - Repliziert Blöcke auf andere Knoten
  - Entfernt lokale Block Replica
  - Re-registrierung oder Shut Down des Knotens
  - Sofortiges Senden eines Block-Reports

Architektur – HDFS Client

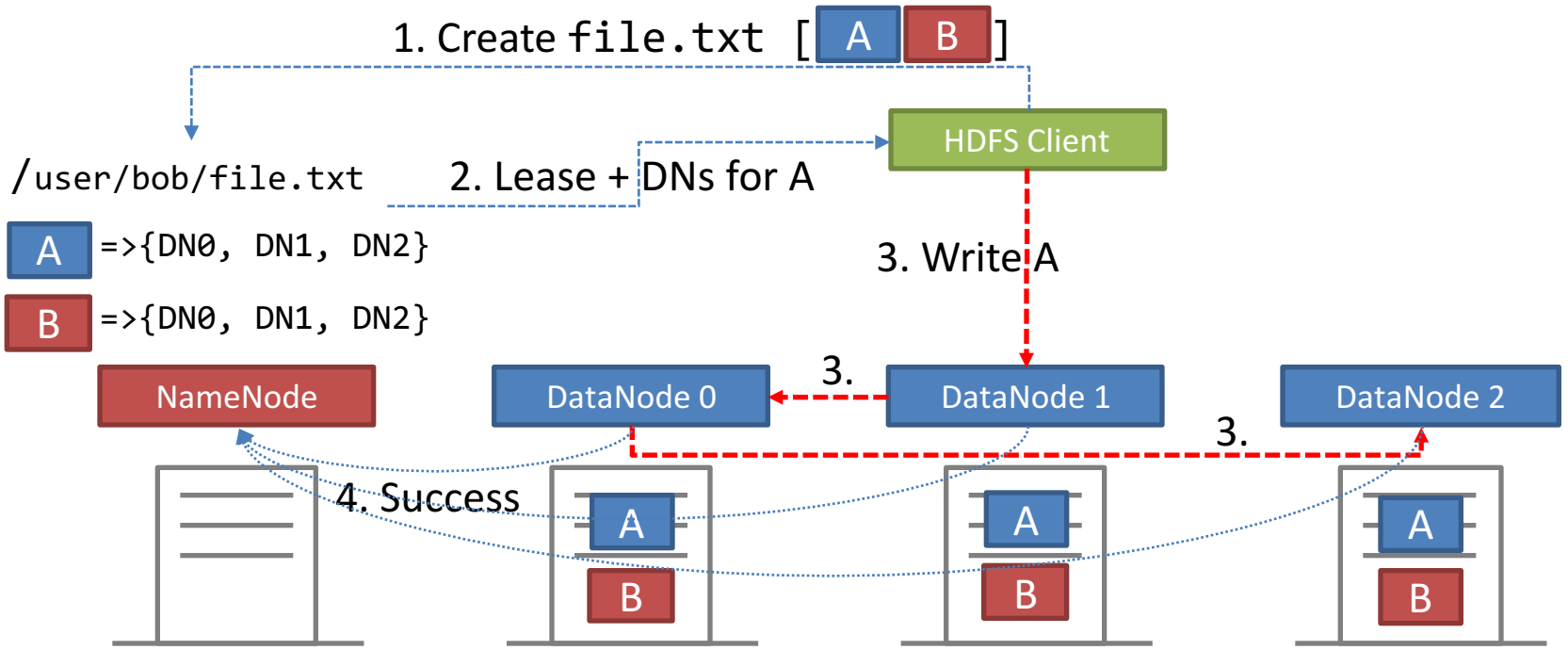


- Bietet Anwendungen Zugriff auf HDFS
- Operationen zum Lesen, Schreiben, Löschen von Dateien (read, write, delete)
- Operationen zur Erzeugung (create) und Löschung (delete) von Ordnern
- Kein Wissen über Replikation und die notwendigen Block-Metadaten
- Übermittelt die Location der Dateiblöcke
  - z.B. von MapReduce zur Verteilung von Tasks verwendet → Auswahl des Servers, der die benötigten Daten vorhält
- Verschiedene Client Implementierungen
  - Java und C API
  - Terminal Client
  - Web Browser

Architektur– Secondary NameNode / Checkpoint Node / Backup Node

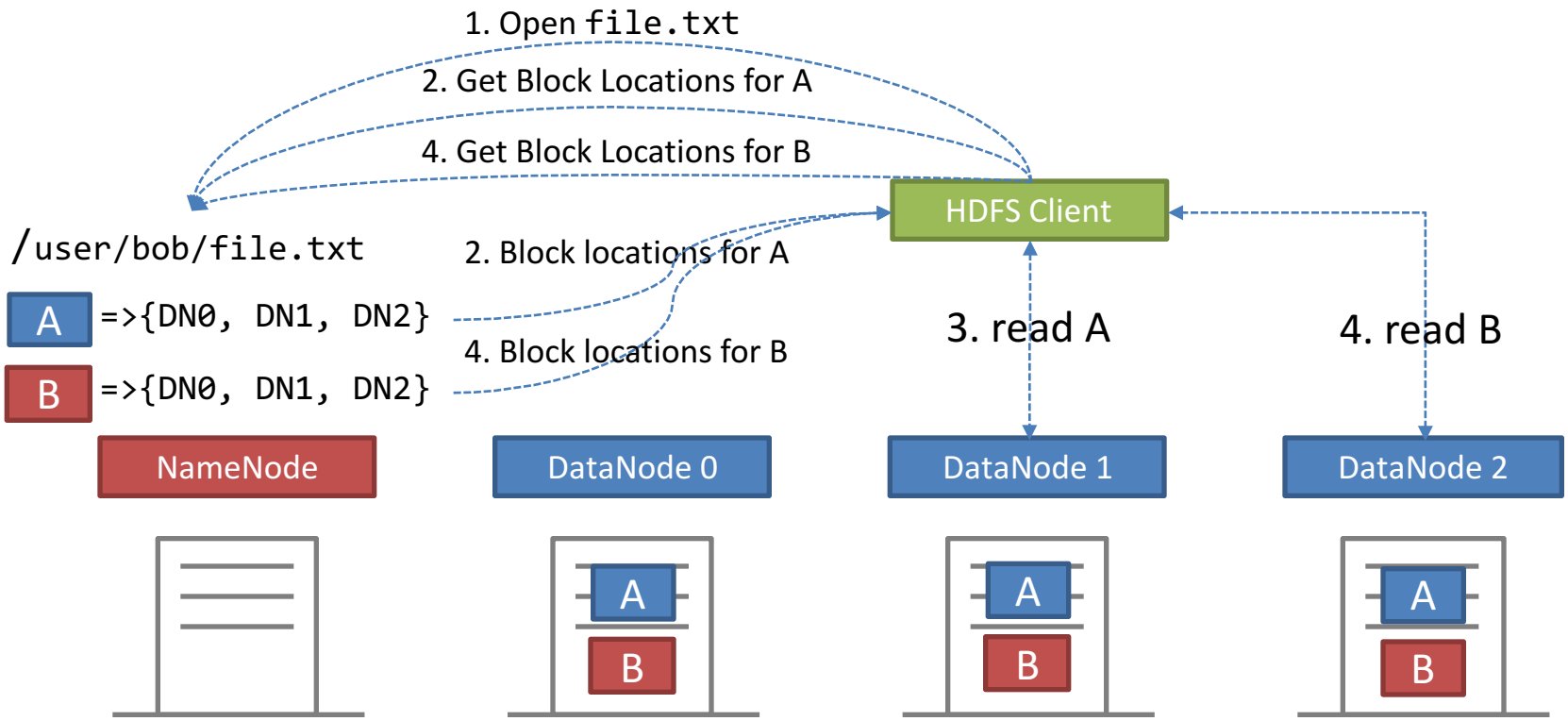
- Secondary NameNode
  - Irreführender Name
  - Periodischer Download des Checkpoint und Journal vom NN
  - Merge von Checkpoint und Journal
  - Sendet neuen Checkpoint zurück an NN
- Checkpoint Node (Hadoop 2)
  - Entspricht Secondary NameNode
- Backup Node
  - Wie Checkpoint Node
  - Führt in-memory, up-to-date Image des Dateisystems
  - Wird durch NN synchronisiert (implementiert die Journal API)
  - „read only" Name Node (enthält keine Block Locations)
  - Vorteil: NN kann ohne persistenten Speicher laufen

HDFS Operationen – Write



1. Erzeugung einer Datei (NN prüft, ob Datei existiert und Nutzer die erforderlichen Rechte hat)
2. NN sendet 'file lease' und DNs, auf die 1. Block geschrieben werden soll (dfs.replication (3))
3. Leitet Write des Blocks A an DN1 → DN0 → DN2 unter Verwendung von **Packets**
4. DNs senden Block-Report an NN (Update Mapping)
5. Write B …

HDFS Operationen – Read

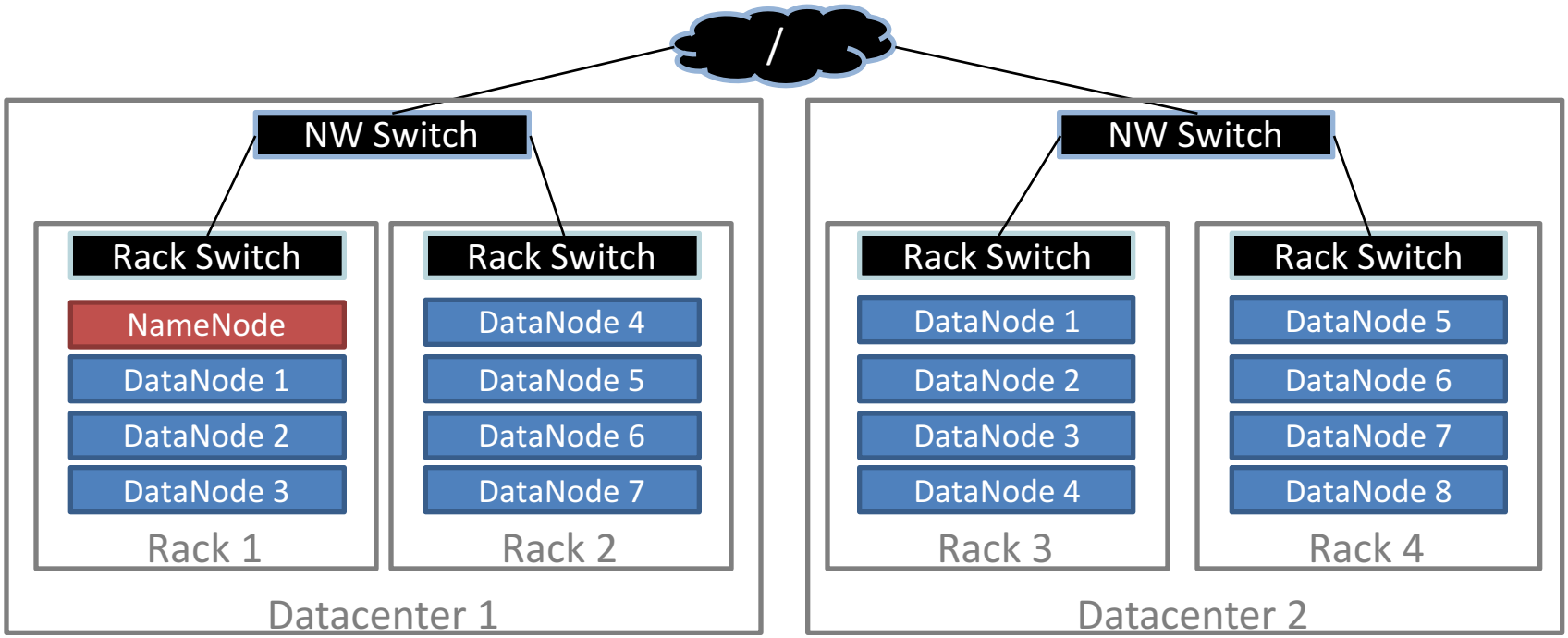


1. Öffnen einer Datei Open für read
2. Erhält Block Locations vom NN sortiert nach **Distanz** zum Client
3. Liest ersten Block (A) vom nächsten, erreichbaren DataNode, verifiziert **Checksum**
4. Liest zweiten Block (B) vom nächsten, erreichbaren DataNode, verifiziert **Checksum**

HDFS Cluster Topologie

Annahme: Distanz zum Vorfahr = 1
Distanz zwischen zwei Nodes = Summe der Distanzen zum kleinsten gemeinsamen Vorfahr

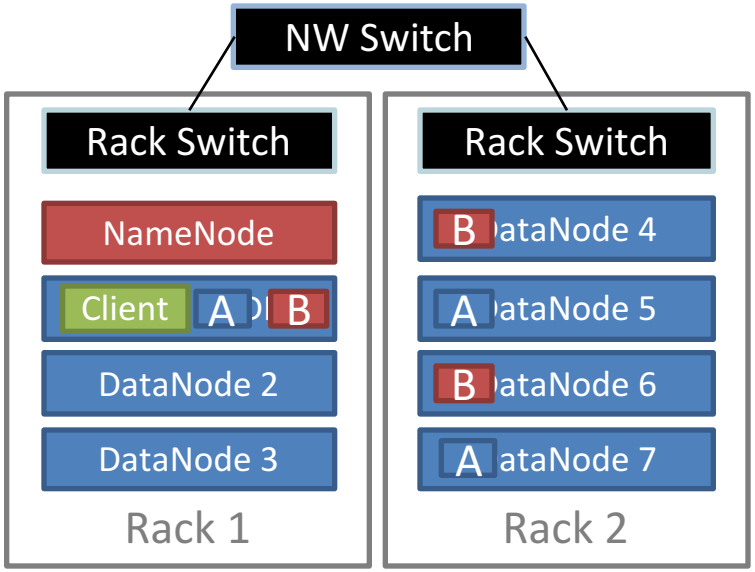
```
distance(/d1/r1/n1, /d1/r1/n1) = 0 (same node)
distance(/d1/r1/n1, /d1/r1/n2) = 2 (different nodes, same rack)
distance(/d1/r1/n1, /d1/r2/n5) = 4 (different racks, same dc)
distance(/d1/r1/n1, /d2/r4/n5) = 6 (different racks, different dc)
```



Block Placement

- Tradeoff zwischen Zuverlässigkeit (reliability), Verfügbarkeit, Schreib- und Lesebandbreite
- Definiert durch Block Placement Policy
  - Default Policy (dfs.replication = 3)
    - 1. Replica auf dem gleichen Knoten wie Client
    - 2. Replica in anderem Rack
    - 3. Replica auf dem gleichen Rack wie 2. aber anderer Knoten
    - 4. bis n-tes Replica Random-Zuordnung (mit Constraints)

1. Create file.txt [A B]
2. Write A
3. NN replies {DN1, DN5, DN7}
4. Write B
5. NN replies {DN1, DN4, DN6}



Block Replication Management und Balancierung

- NameNode verwaltet Replikationsstatus der Blöcke (Block Reports)
  - „over-replicated" Block
    - Rack wird re-aktiviert nach Netzwerkpartitionierung; Änderung von dfs.replication
    - NN wählt zu entfernende Replica aus (versucht nicht die Anzahl der Racks zu minimieren, bevorzugt „sehr volle" DNs)
  - „under-replicated" Block
    - Disk / Server / Rack Ausfall; Änderung von dfs.replication
    - Block wird Replication Priority Queue hinzugefügt (nur 1 Replica = höchste Priorität)
    - Queue wird periodisch gescannt und Blöcke werden entsprechend Block Placement Policy verteilt
  - Alle Replicas sind auf einem Rack (Rack Ausfall)
    - Behandlung des Blocks wie „under-replicated"
- Balancer Tool
  - Administratives Tool, das Über- und Unterauslastung der DNs ermittelt und zum Erreichen einer guten Balance die Blöcke umverteilt