HSLU Hochschule Luzern

Verteilte Systeme und Komponenten

# Skalierung und Verteilung

Martin Bättig

**Letzte Aktualisierung: 23. November 2022**

FH Zentralschweiz

# Inhalt

- Grundlagen der Skalierung und Verteilung
- Lastverteilung mittels Reverse-Proxys
- Lastverteilung mittels In-Memory Datagrids
- Zusammenfassung

# Lernziele

- Sie kennen die Grundlagen der Skalierung und Verteilung.
- Sie wissen was ein Reverse-Proxy ist und wie dieser zur Lastverteilung eingesetzt werden kann.
- Sie verstehen, wie man mittels Kombination eines Load-Balancers und Replikas eine Skalierung erzielt.
- Sie können nachvollziehen wie ein In-Memory Data Grid (IMDG) Daten speichert und in einem Cluster verteilt und wissen in den Grundzügen, was passiert, wenn im Cluster ein neuer Knoten hinzukommt oder wegfällt.
- Sie können einfache Code-Beispiele am Beispiel von HazelCast nachvollziehen und selbst welche mit wenigen Zeilen schreiben.

# Grundlagen der Skalierung und Verteilung

# Quiz zum Aufwärmen

Angenommen Sie hätten vier Serversysteme zur Verfügung mit jeweils:

- 32GB Arbeitsspeicher (Zugriffszeiten im ns-Bereich).
- 1TB Harddisk (Zugriffszeiten im ms-Bereich).
- Jeweils identische und schnelle Netzwerkanbindung mit 10GBit/s.

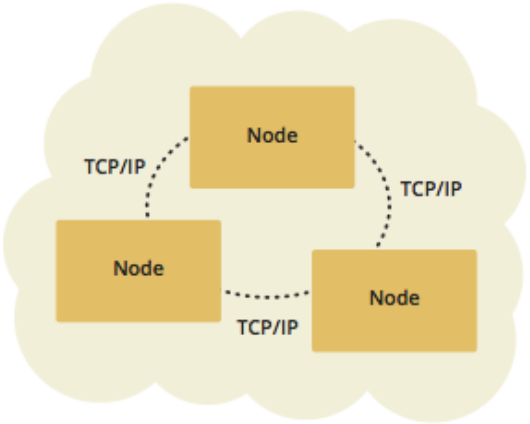
Sie möchten mittels diesen Systemen Bilddaten ausliefern:

- Ein Bild ist maximal 10M gross.
- Sie haben im Total 100GB an Bilddaten.

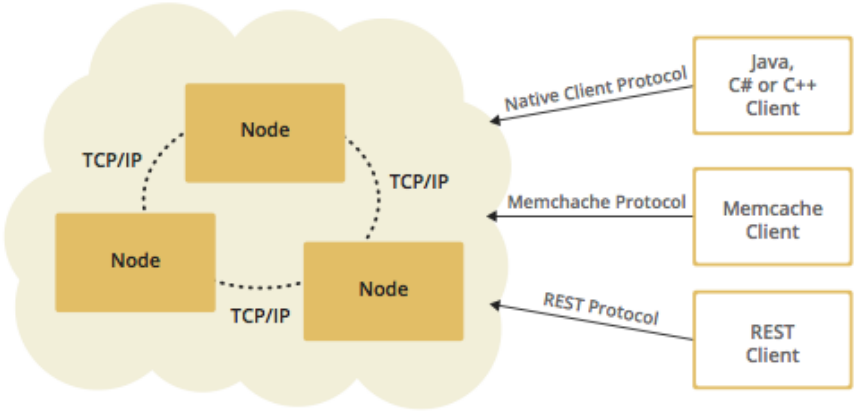
**Frage:** Wie können Sie diese Systeme programmieren, sodass Sie jedes Bild mit möglichst mit kleiner Latenz ausliefern können?

# Topologie verteilter Systeme

- **Embedded**: Für Anwendungen wo asynchrone Ausführung von Task oder Hochleistungs-Computing gefragt sind. Knoten enthalten in diesem Typ sowohl die Anwendung als auch die Daten.




- **Client / Server**: Für Cluster von Server-Knoten, die erstellt und skaliert werden können. Clients kommunizieren mit diesen Serverknoten, um auf die Daten zu zugreifen. Es gibt native Clients (Java, .NET und C++), Memcache-Clients und REST-Clients.



# Skalierung verteilter Systeme

**Zustandslose Systeme** (alle Anfragen sind unabhängig):

- Mehrere Instanzen einer Serveranwendung starten.
- Jede Anfrage je nach Auslastung an die Instanzen verteilen.

⇨ Einfache Lastverteilung.

**Zustandsbehaftete Systeme mit temporären Daten:**

- Mehrere Instanzen einer Serveranwendung starten.
- Erste Anfrage einer Session je nach Auslastung an eine Instanz N verteilen. Folgeanfragen jeweils wieder an die gleiche Instanz N leiten.

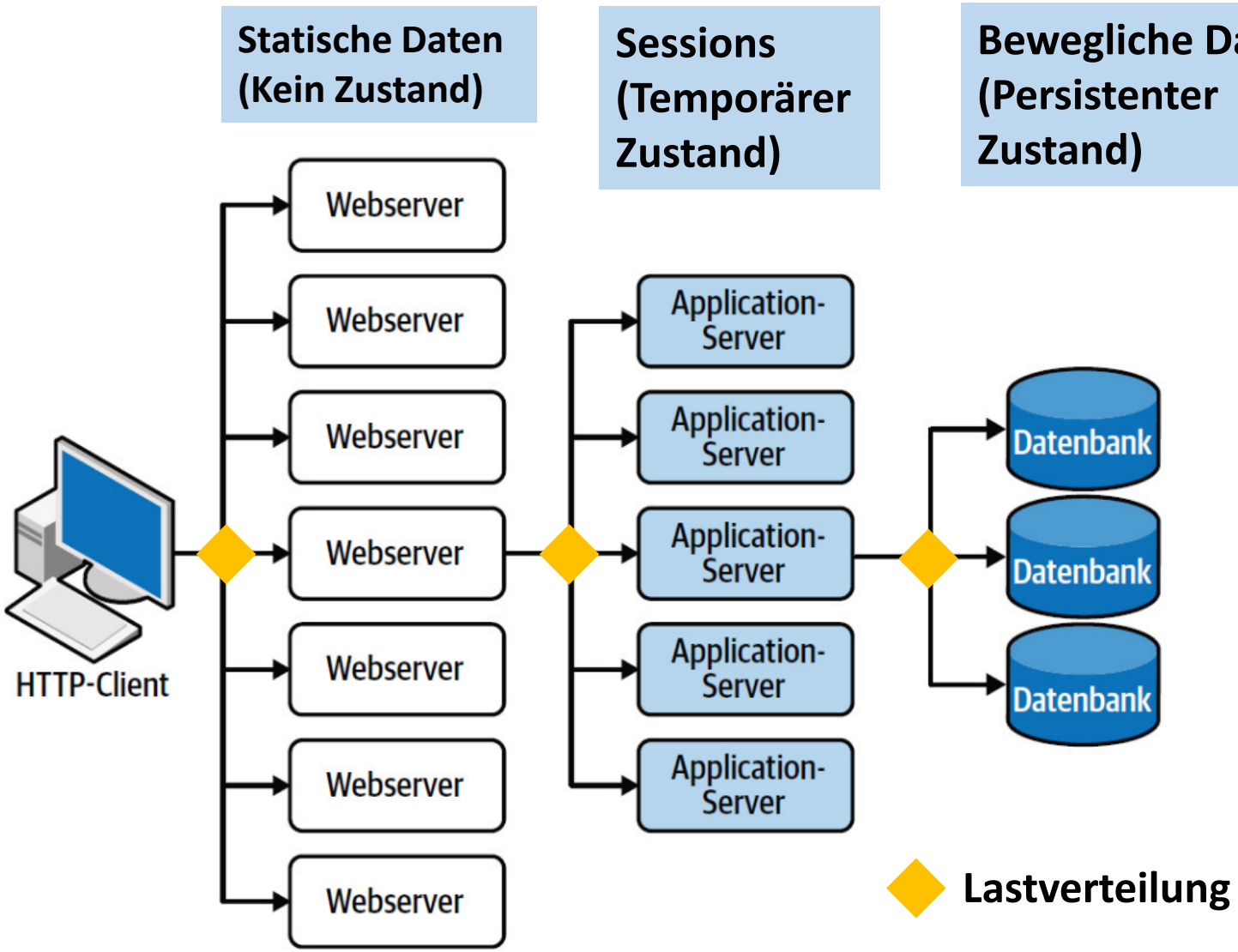
⇨ Komplexere Lastverteilung oft mittels Inspektion der Kommunikation.

**Zustandsbehaftete Systeme mit persistenten Daten:**

- Daten replizieren (ggf. Partitionieren, d.h. zw. Instanzen aufteilen).
- Anfragen an diejenigen Instanzen der Serveranwendung verteilen, welche entsprechende Daten vorhält.

⇨ Komplexeste Lastverteilung (read vs. write, Konsistenz der Replikas, etc.)

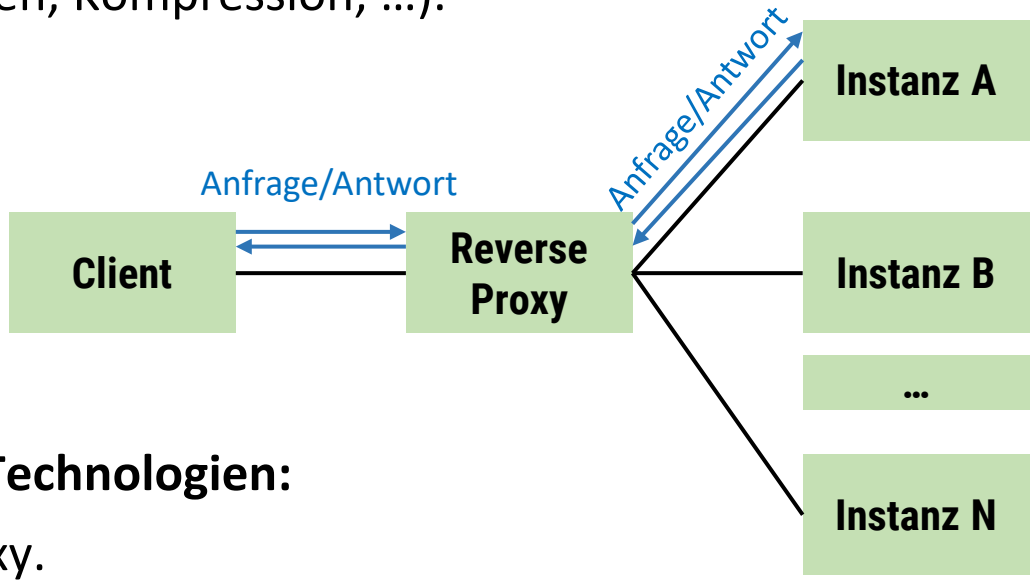
# Beispiel: Skalierung einer klassischen Webapplikation



# Lastverteilung mittels Reverse-Proxys

# Reverse-Proxy

- Vorgeschaltete Middleware, welche Anfragen auf verschiedene Instanzen einer Serveranwendung verteilt.
- **Load-Balancing:** Last wird auf mehrere Instanzen verteilt, sodass keine Instanz überlastet ist, solange noch andere Instanzen Kapazität haben.
- Oft weitere Funktionalität z.B. Sicherheit (Verschlüsselung, Monitoring, Metriken, Kompression, …).




**Beispiel Technologien:**

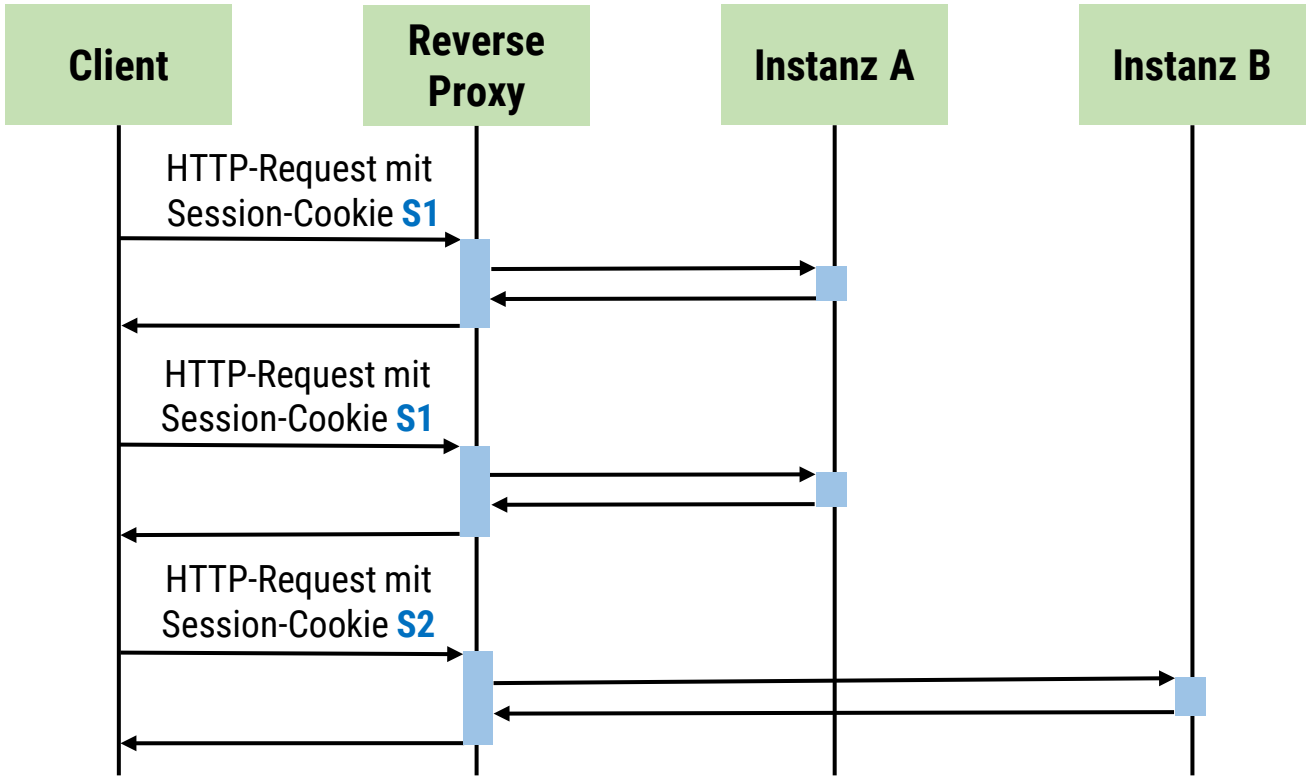
- HAProxy.
- Webserver wie Apache / Nginx
- Træfik

# Lastverteilungsmethoden (Auswahl)

- **Round-Robin:** (Der Reihe nach) Erstelle eine Liste mit allen Instanzen. Die Liste wird abgearbeitet und jede Instanz in dieser Liste erhält eine Anfrage. Hat die letzte Instanz eine Anfrage erhalten, beginnt der Proxy von vorne.
  Gut bei homogener Last der Anfragen und homogener Systemausstattung.
- **Anzahl bestehender Verbindungen:** Leite Anfrage zur Instanz mit der geringsten Anzahl Verbindungen weiter.
  Gut für langdauernde TCP-Verbindungen.
- **Hash:** Anwendung einer Hash-Funktion auf IP-Adresse des Clients um die Zielinstanz zu bestimmen. Alle Anfragen einer IP-Adresse werden an die identische Instanz weitergeleitet.
  Einfache Möglichkeit Requests an gleichen Server zu leiten (benötigt jedoch stabile Client-IPs).

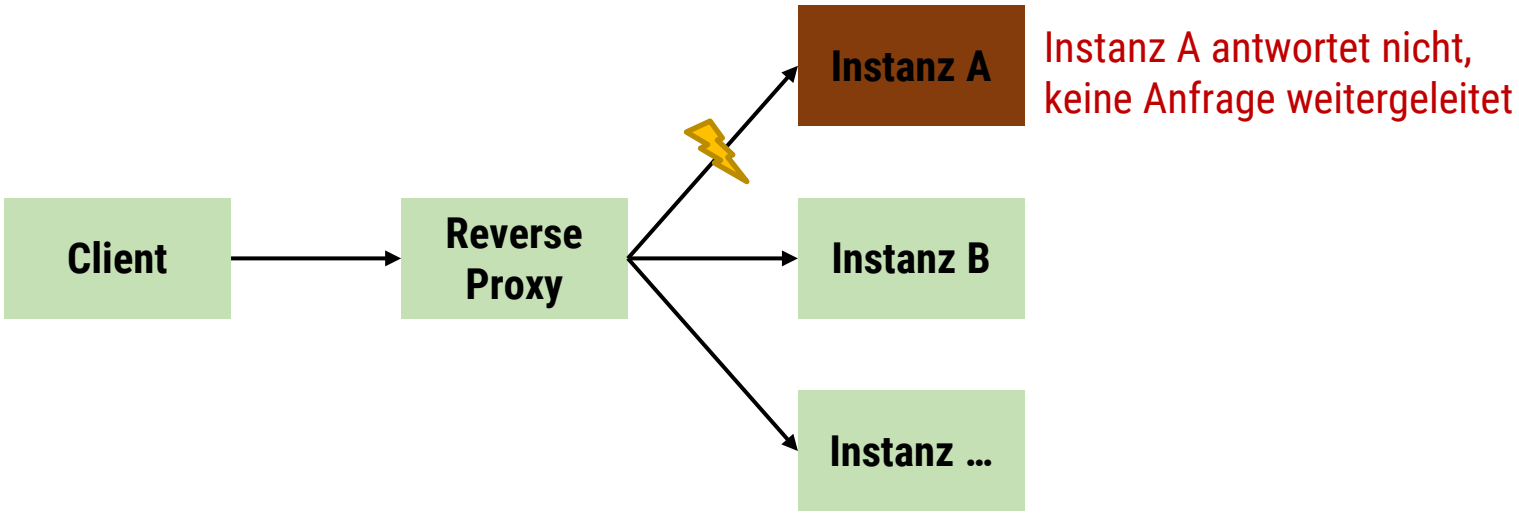
# Zusammenspiel mit Serverapplikation

Sollen mehrere aufeinander folgende Requests unabhängig der TCP-Verbindung zur gleichen Serverinstanz geleitet werden, muss der Reverse-Proxy das Protokoll inspizieren können (Teilverständnis).



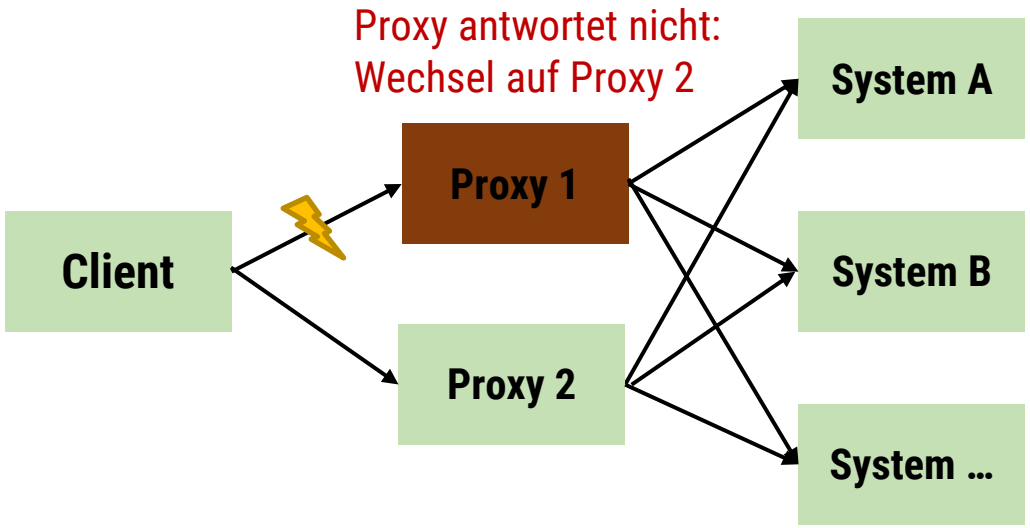
# Ausfallsicherheit auf Seite Serverinstanz

- Load-Balancer testen mittels Health-Checks, ob eine Instanz funktioniert:
  - Keine Weiterleitung von Anfragen an Instanzen, welche nicht antworten.
- Typische Health-Checks:
  - Steht TCP-Verbindung (Transport-Schicht).
  - Beliebiger HTTP-Request (Applikations-Schicht).
  - Ausführen eines Agents (Hilfsprogramm) auf Serverinstanz.



# Ausfallsicherheit auf Seite Proxy

- Einzelner Reverse-Proxy wäre Single-Point of Failure.
- Mindestens zwei Instanzen für Ausfallsicherheit benötigt:
  - Kombination mit Hochverfügbarkeitslösungen (z.B. keepalived) und Floating-IPs (IP, welche auf verschiedene Ziele geroutet werden kann).
  - Bei Ausfall eines Proxys wird die IP gewechselt.



# Beispiel: HAProxy

**Eckdaten:**


- Beziehen unter: https://www.haproxy.org
- Open-Source und kommerzielle Version.
- Architektur: Asynchroner IOs in Kombination mit mehreren Threads.
- Kommerzielle Version hat diverse Erweiterungen (z.B. Routing nach GeoIP oder Traffic-Mirroring).

**Einfacher Aufbau:**

- Einzelnes Binary: haproxy
- mit Konfiguration: Z.B. haproxy.conf

**Start des Proxys:**

```
haproxy -V -f haproxy.conf
```

(in Verbose-Mode mit Konfigurationsfile haproxy.conf):

# HAProxy: Konfiguration

Einfache Beispielkonfiguration für einen TCP-Server mit zwei Instanzen.

```
global

defaults
    mode tcp
    balance roundrobin
    timeout connect 5000ms
    timeout client 50000ms
    timeout server 50000ms

frontend http-in
     bind *:4000
     default_backend servers

backend servers
     server server1 127.0.0.1:8000 check
     server server2 127.0.0.1:8001 check
```

**Protokoll:** tcp | http | …

**Lastverteilung:**
roundrobin | leastconn | source | …

**Timeouts:**
- Verbindungsaufbau (connect)
- Client/Server Inaktivität (Client / Server)

**Frontend:** Angabe des Listener-Interface für die Clients sowie des Ziel-Backends.

**Backend:** Liste von Instanzen der Serverapplikation.

# Klassenraumübung: Load-Balancing Hands-on

Übung mittels Online-Programmierumgebung:

- https://replit.com/@mbaettig/Load-Balancing
  - **Kein HSLU-Dienst (benötigt separates Konto), erstellen Sie einen Fork.**

**Ziel:** Machen Sie eigene Versuche mit Load-Balancing.

**Vorgehen:** Starten Sie vier Shells:

- **Shell 1:** `haproxy -V -f haproxy.conf`
- **Shell 2:** `java DaytimeServer srv1 127.0.0.1 8000`
- **Shell 3:** `java DaytimeServer srv2 127.0.0.1 8001`
- **Shell 4:** `java DaytimeClient 127.0.0.1 1300`

**Hinweise:**

- Stoppen jeweils mit Ctrl-C.
- Zum Neuladen der Konfiguration muss HAProxy neugestartet werden.

# Klassenraumübung: Load-Balancing Hands-on (forts.)

**Machen Sie folgende Experimente und notieren Sie Ihre Beobachtungen:**

- Führen Sie die Zeitabfrage (DaytimeClient) mehrfach aus.
- Mehrere Zeitabfragen mit anderer Lastverteilung: source und/oder leastconn.
- Machen Sie mehrere Zeitabfragen während:
  - Ein oder beide Server gestoppt sind.
  - Einer oder beide Server wieder gestartet sind.

# In-Memory Datagrids

# Idee: Transparent verteilte statt lokale Maps

Beispiel mit einer Collection Map für die In-Memory Speicherung von Daten…

```
import java.util.Map;
import java.util.HashMap;

Map<Integer, String> map = new HashMap<>();
map.put(1, "value");
String result = map.get(1);
```

# Verteilte In-Memory Map

…wenn die In-Memory Speicherung von Daten parallel und verteilt sein soll (mit gemeinsamer Ressource)…

```
import com.hazelcast.core.HazelcastInstance;
import com.hazelcast.core.Hazelcast;
import java.util.Map;

HazelcastInstance client = Hazelcast.newHazelcastInstance();
Map<Integer, String> map = client.getMap("mymap");
map.put(1, "value");
String result = map.get(1);
```

# Was können In-Memory Datagrids?

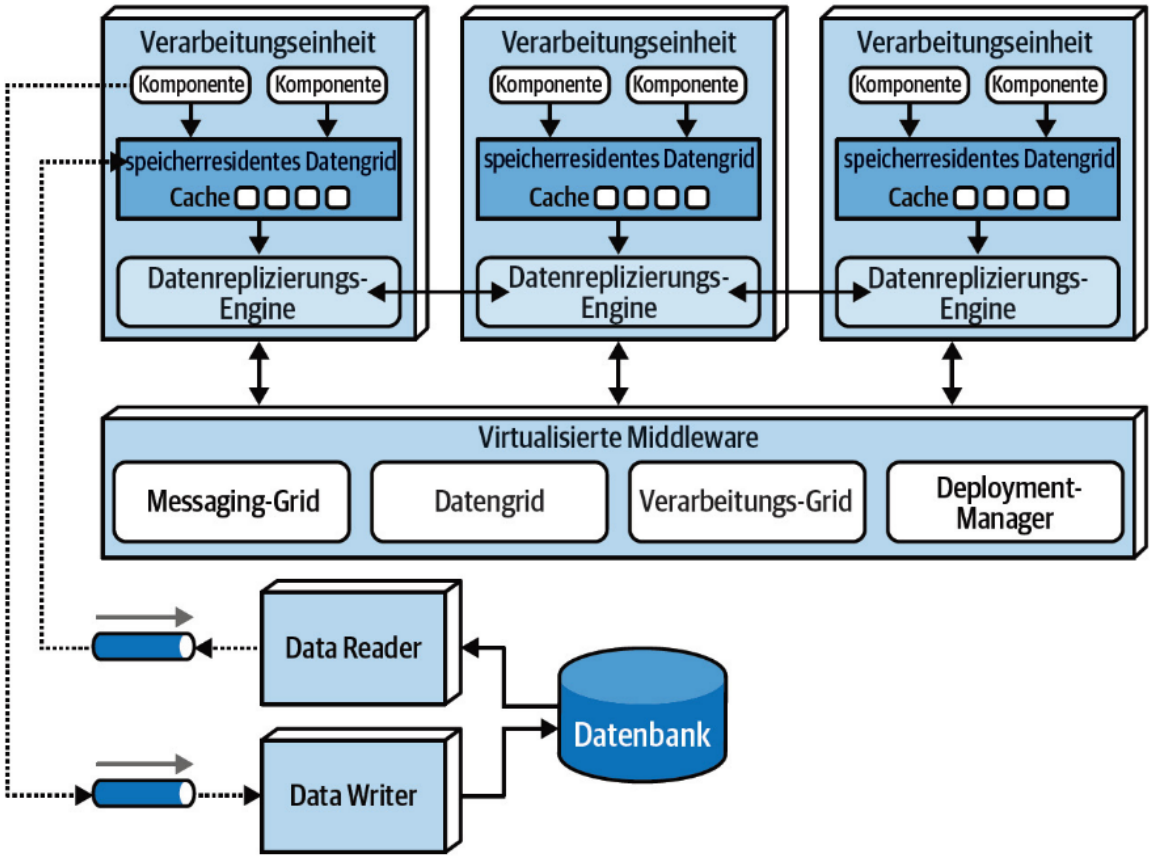
- Applikationen skalieren
- Daten über Cluster verteilen
- Daten partitionieren
- Nachrichten senden und empfangen
- Lasten verteilen
- Parallele Tasks verarbeiten
- …

**Implementationen von In-Memory Datagrid-Technologien:**

- HazelCast: https://hazelcast.com
- Apache Ignite https://ignite.apache.org
- Oracle Coherence https://oreil.ly/XOUJL

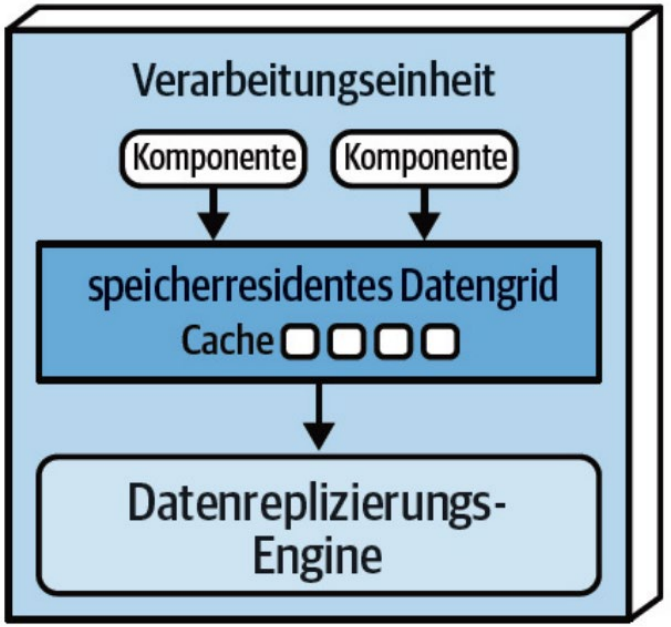
# Architektur auf grosser Flughöhe

- **Verarbeitungseinheiten:** Halten Teile der Daten im Hauptspeicher bereit.
- **Datenbank:** Speichert alle Daten dauerhaft.
- **Middleware:** Kommunikation zwischen Verarbeitungseinheiten.

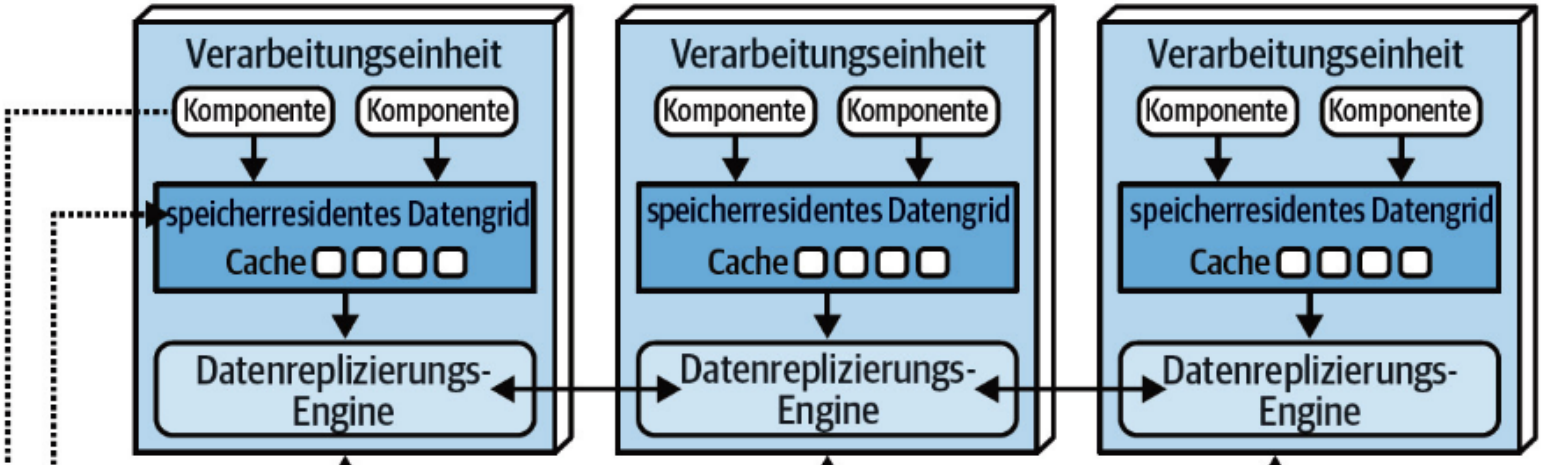


# Verarbeitungseinheiten (Cluster node)

- Embedded-Topologie: Logik und Daten.
- Daten liegen im RAM-Speicher des Cluster Nodes:
  - schneller als Disk optimierte Datenbanken.
  - verbesserte Reaktionszeit bei kritischen Anwendungen.



# Redundanz, Skalierbarkeit und Elastizität



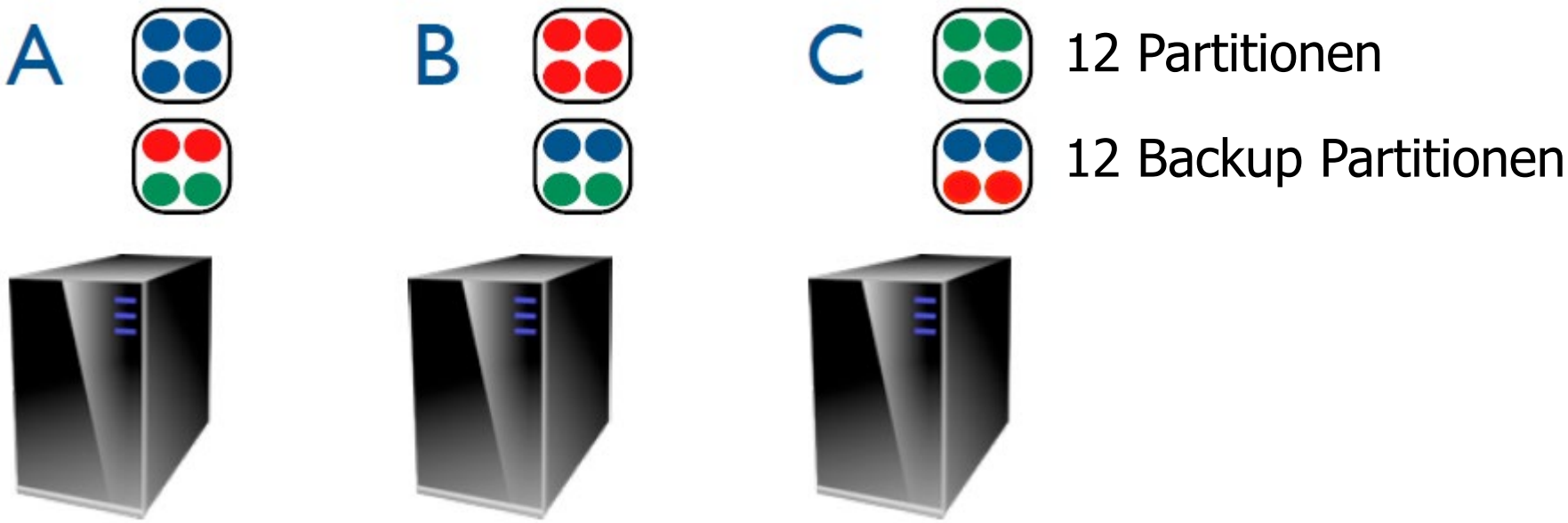

- **Skalierbarkeit:** über verschiedene Cluster-Nodes verteilt.
- **Redundanz:** mehrere Kopien der Daten in verschiedenen Cluster-Nodes
- **Elastizität:** Cluster-Nodes können im Betrieb hinzugefügt oder entfernt werden

# Datenpartitionierung in einem Cluster

- Fixe Anzahl von Partitionen
- Für jede Partition gibt es einen Schlüssel.

```
partitionId = hash(keyData) % PARTITION_COUNT
```

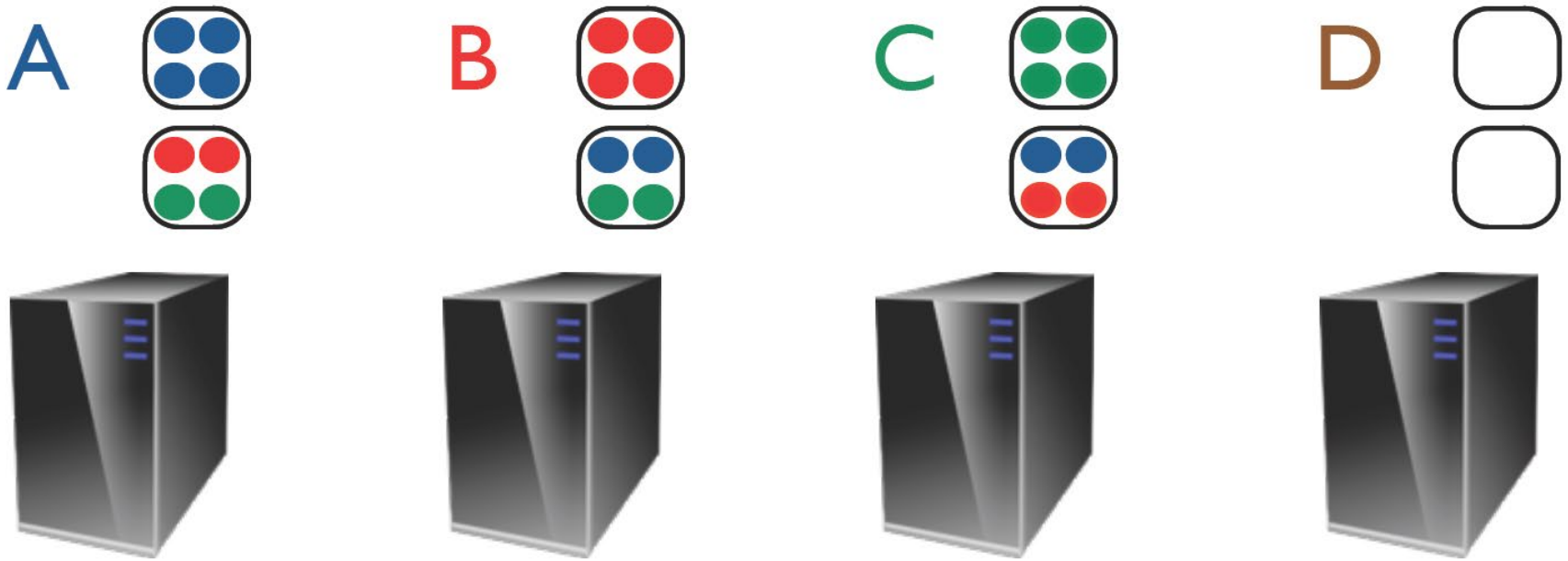
- Alle Partitionen sind möglichst gleichverteilt über den Cluster
  - Backups / Redundanz



# Funktionsweise von In-Memory Datagrids

Neuer Knoten (D) kommt hinzu…

- enthält noch keine Partitionen

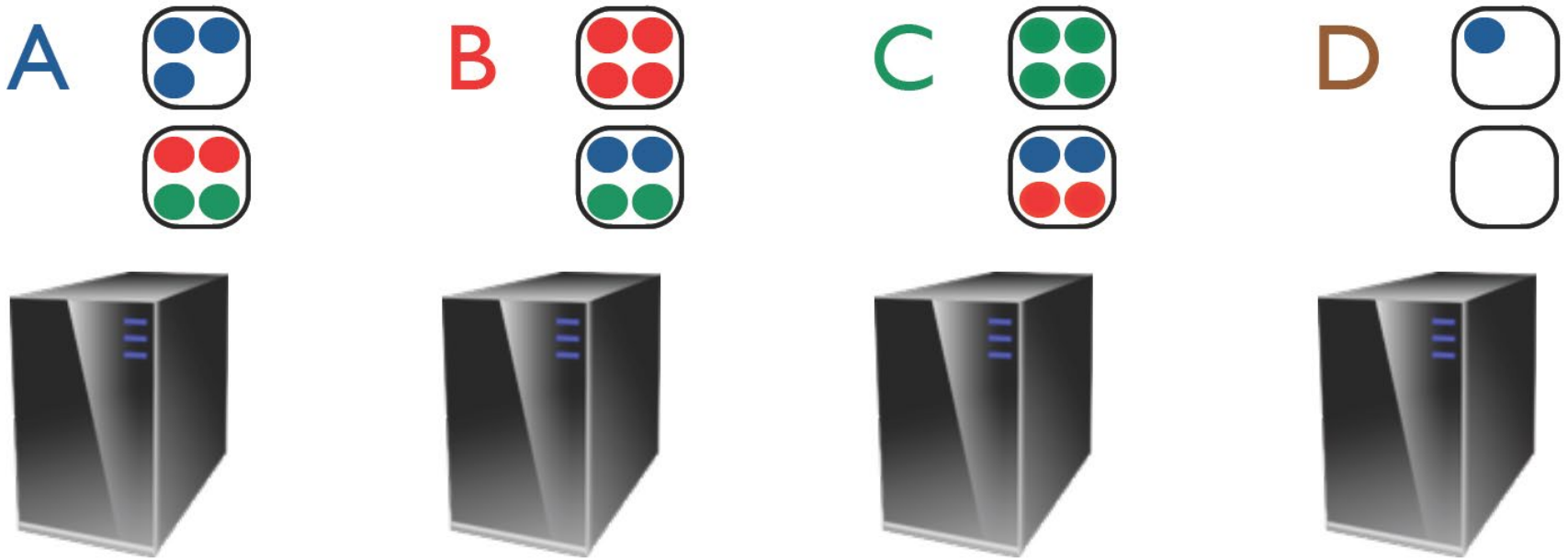



Die folgenden Aktionen müssen in der Praxis nicht in dieser Reihenfolge ablaufen.

# Funktionsweise von In-Memory Datagrids

Migration…

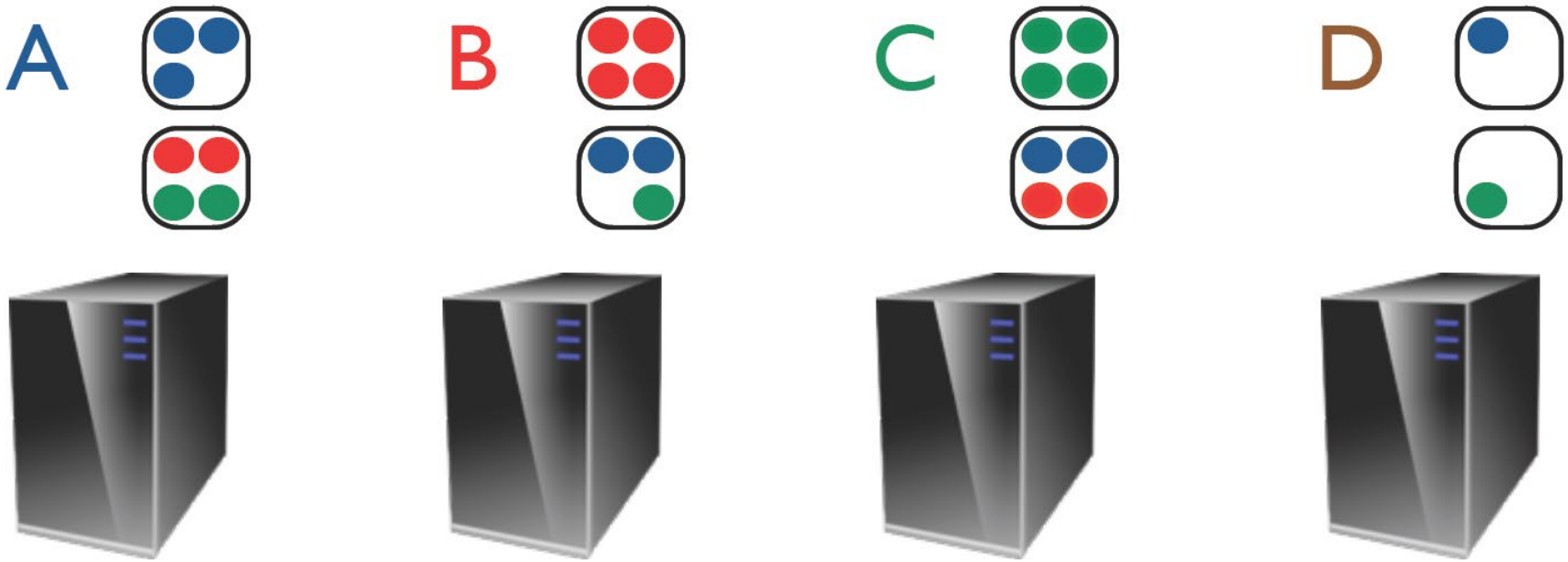
- Knoten D übernimmt Partition von Knoten A



# Funktionsweise von In-Memory Datagrids

Migration…

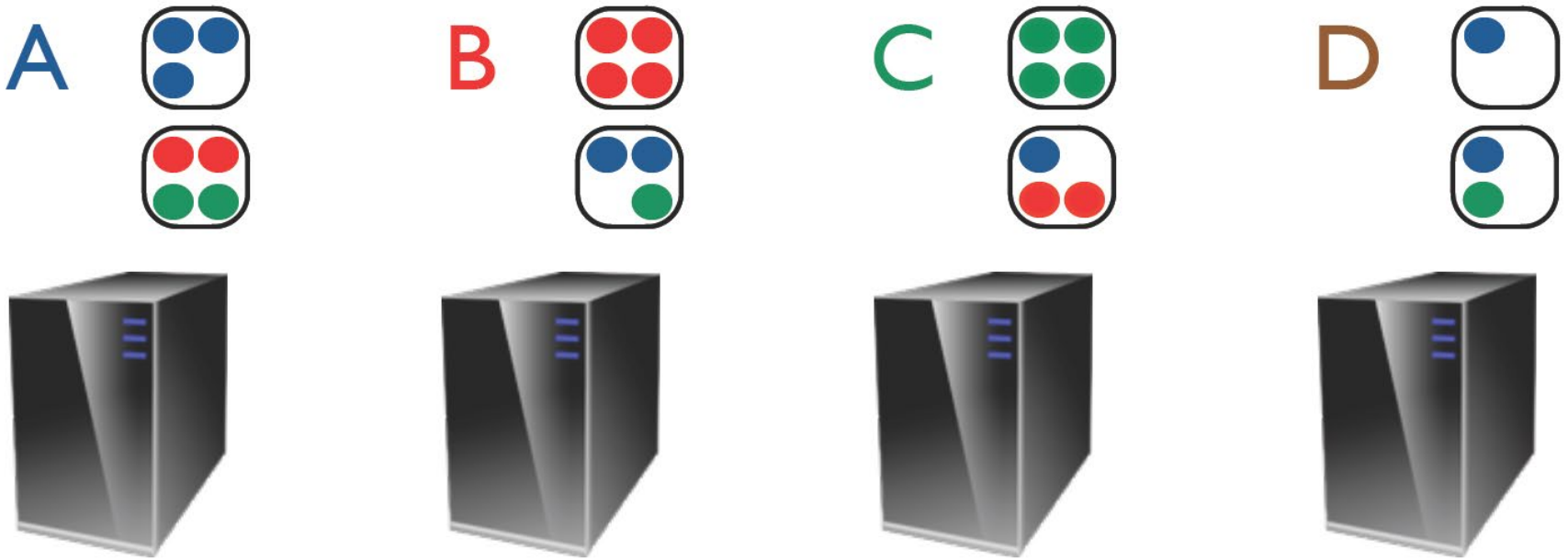
- Knoten D übernimmt Backup Partition C von Knoten B



# Funktionsweise von In-Memory Datagrids

Migration…

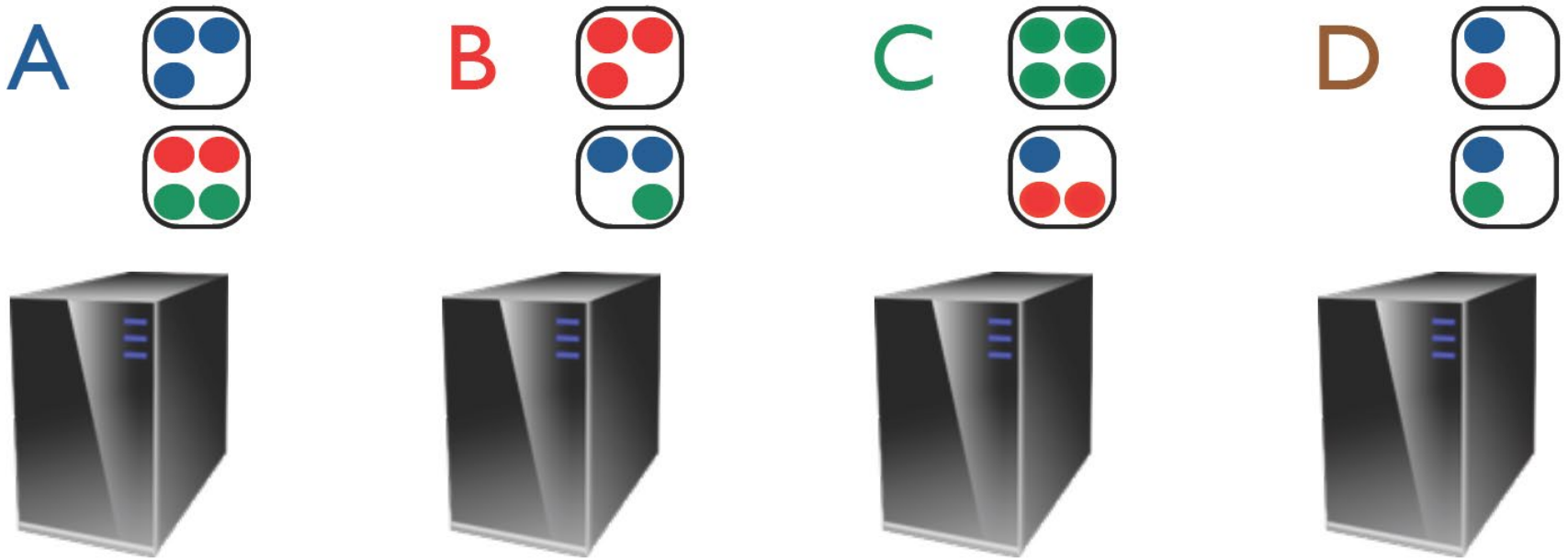
- Knoten D übernimmt Backup Partition A von Knoten C



# Funktionsweise von In-Memory Datagrids

Migration…

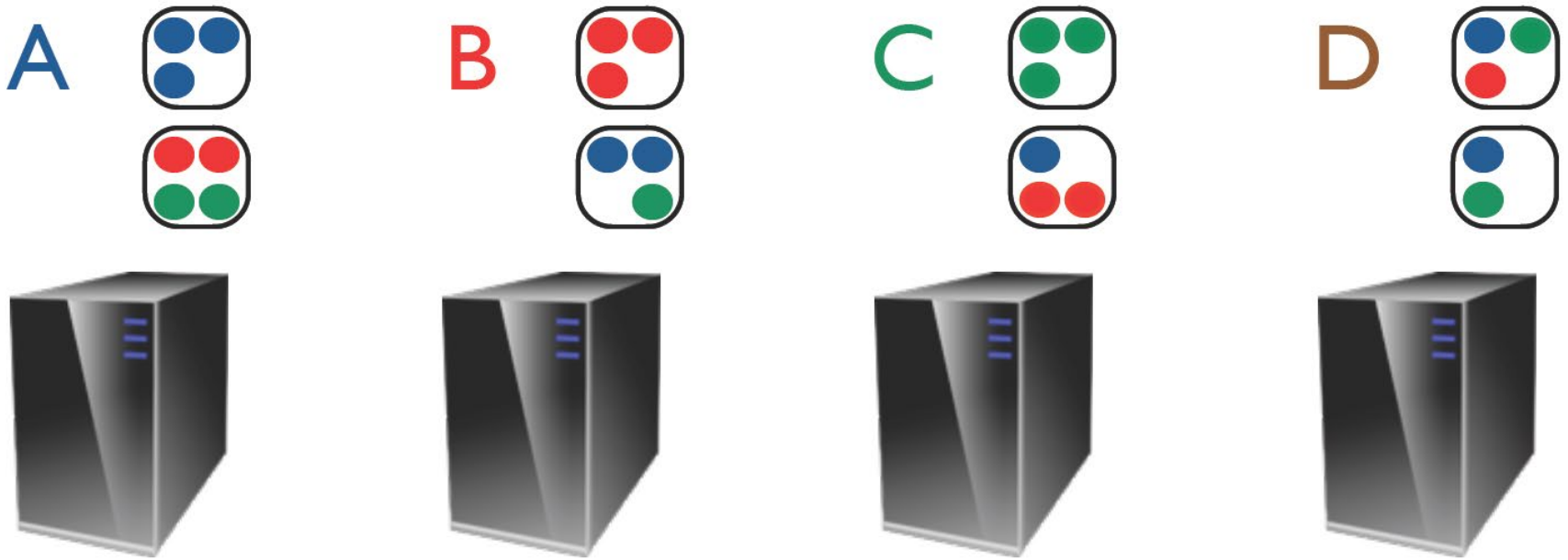
- Knoten D übernimmt Partition von Knoten B



# Wie funktioniert Hazelcast?

Migration…

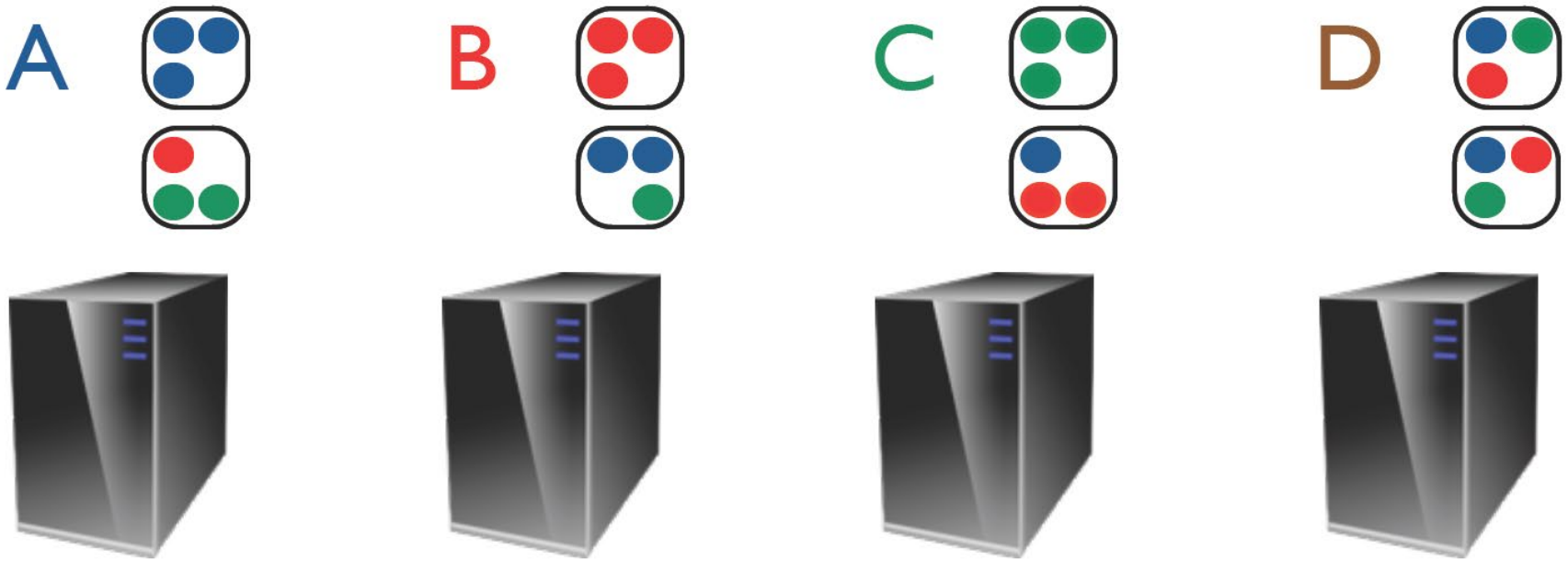
- Knoten D übernimmt Partition von Knoten C



# Wie funktioniert Hazelcast?

Migration…

- Knoten D übernimmt Backup Partition B von Knoten A

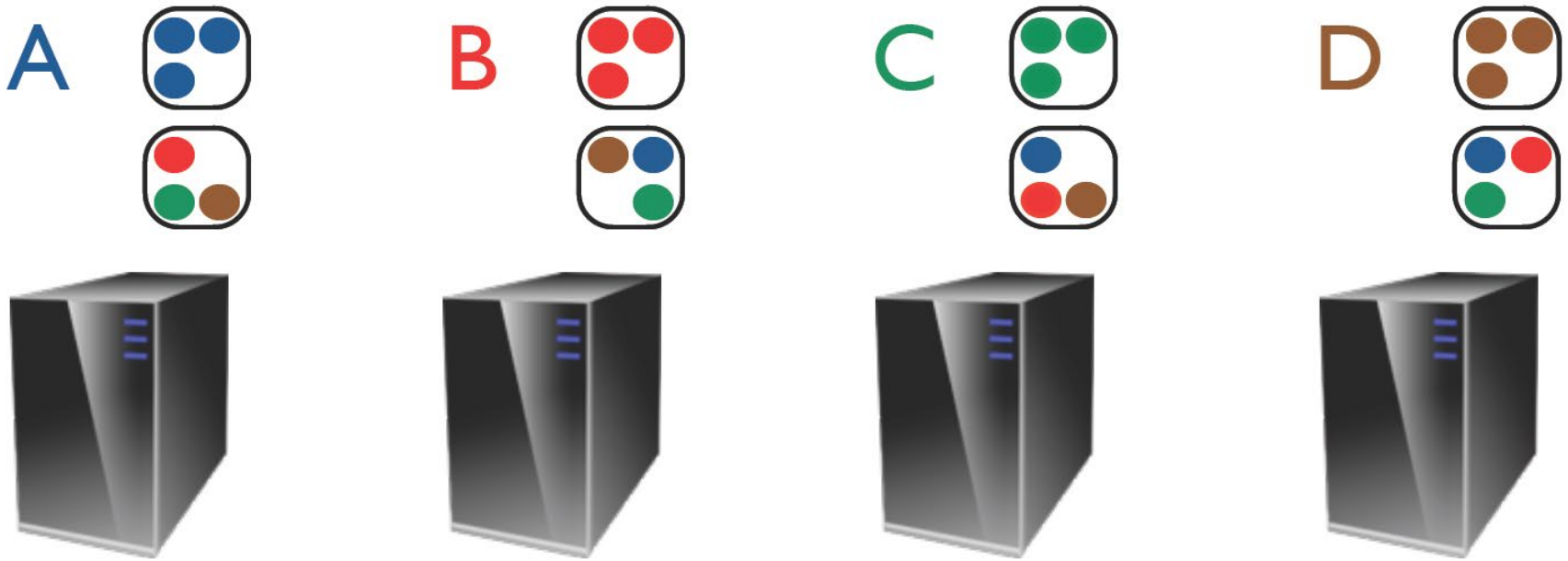



Knoten D besitzt nun Backup Partitionen der andern Knoten.

# Wie funktioniert Hazelcast?

Migration komplett – 12 Partitionen

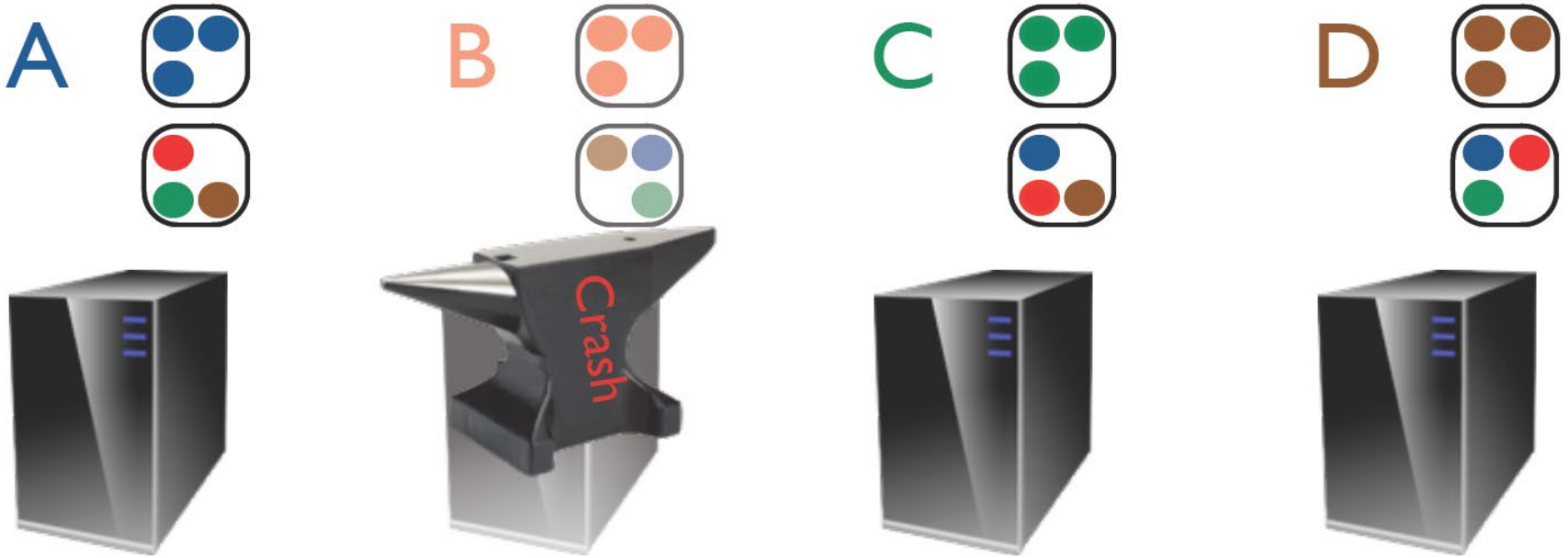
- Knoten D besitzt nun eigene Partitionen (ehemals Partitionen der andern Knoten – Besitzwechsel)




Alle andern Knoten besitzen ein Backup von Knoten D.

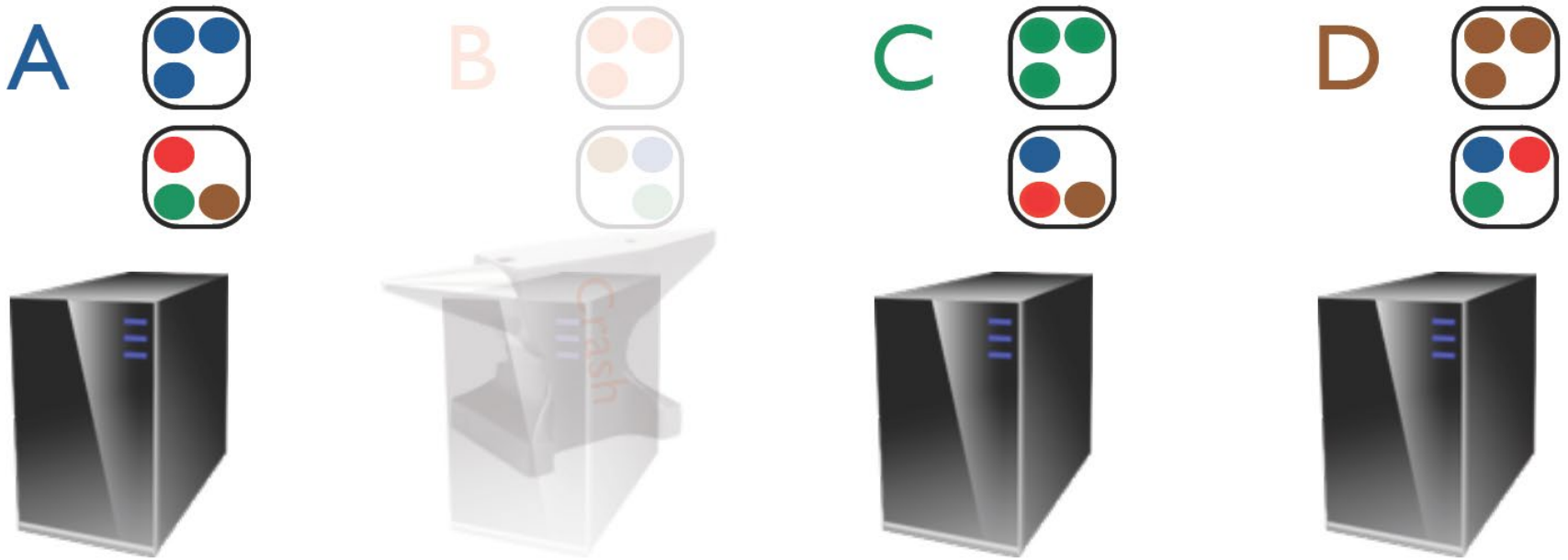
# Wie funktioniert Hazelcast?

Knoten B stürzt ab…



# Wie funktioniert Hazelcast?

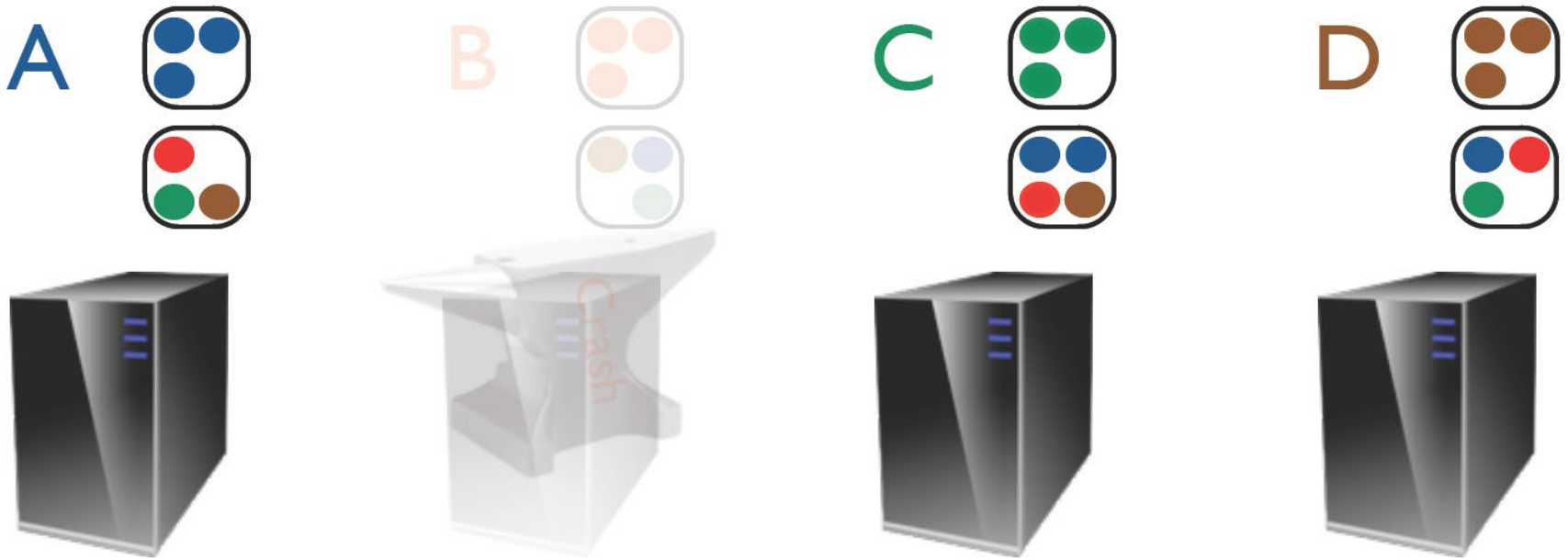
Wiederherstellung mit Hilfe der Backups…



# Wie funktioniert Hazelcast?

Backup anlegen…

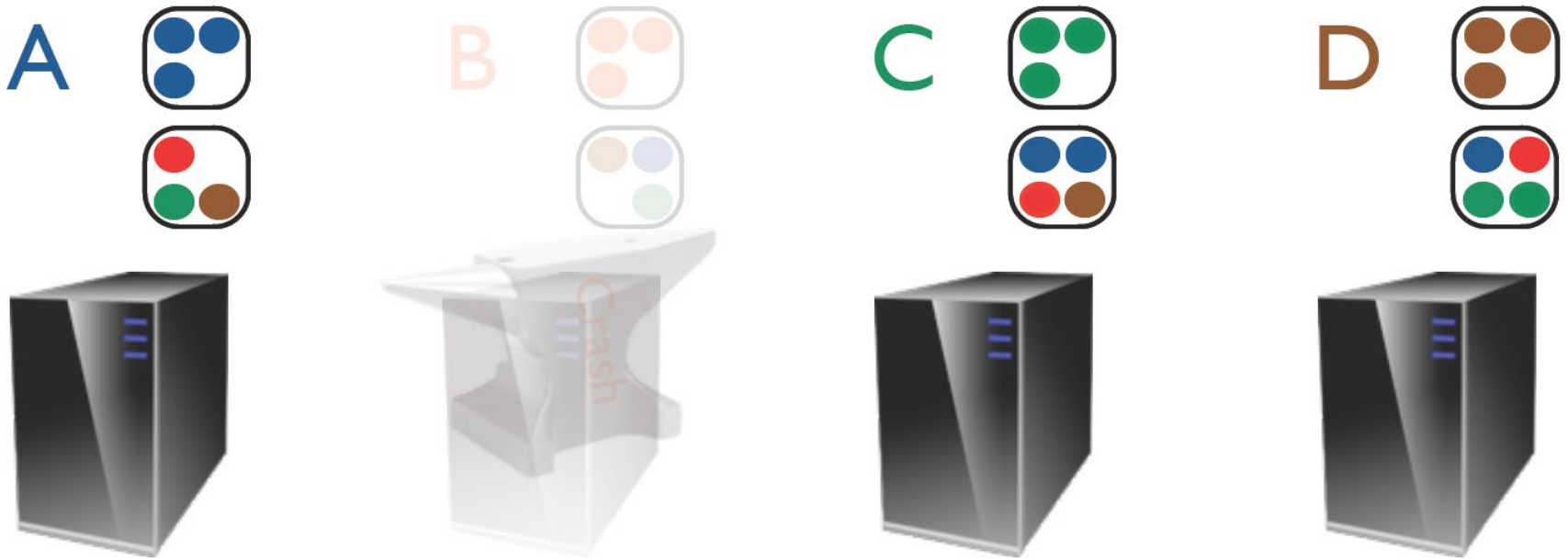
- Knoten C kopiert Partition von A (die auf Knoten B war)



# Wie funktioniert Hazelcast?

Backup anlegen…

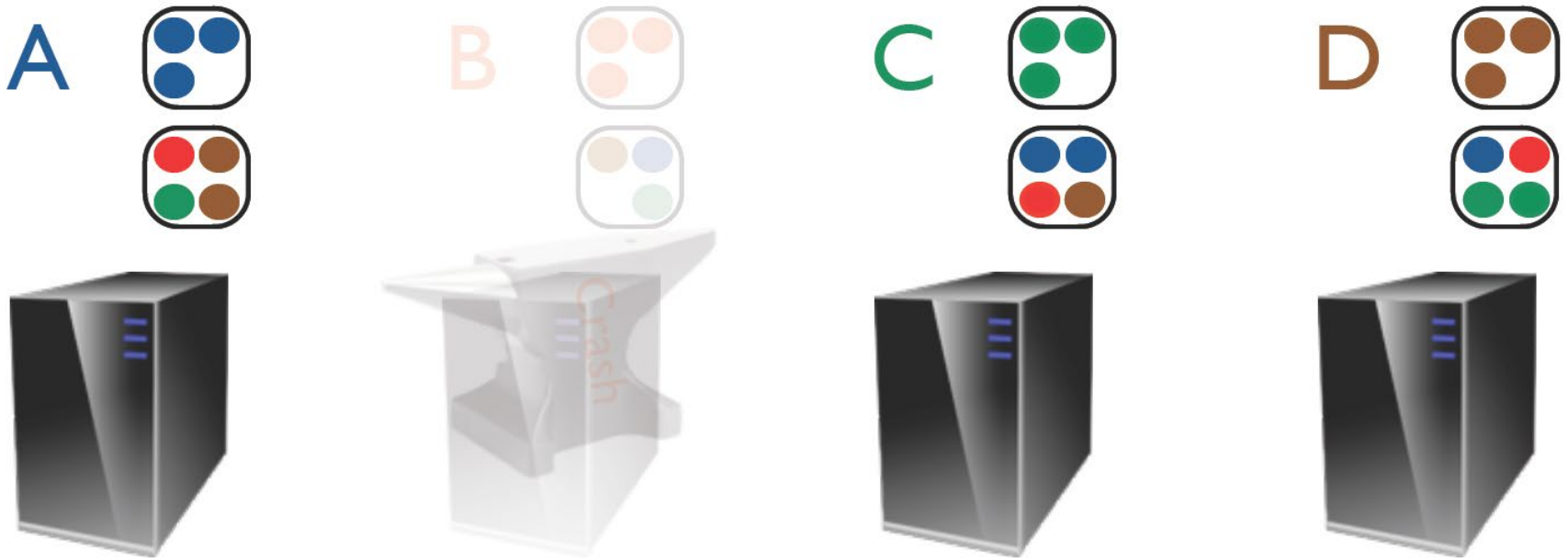
- Knoten D kopiert Partition von C (die auf Knoten B war)



# Wie funktioniert Hazelcast?

Backup anlegen…

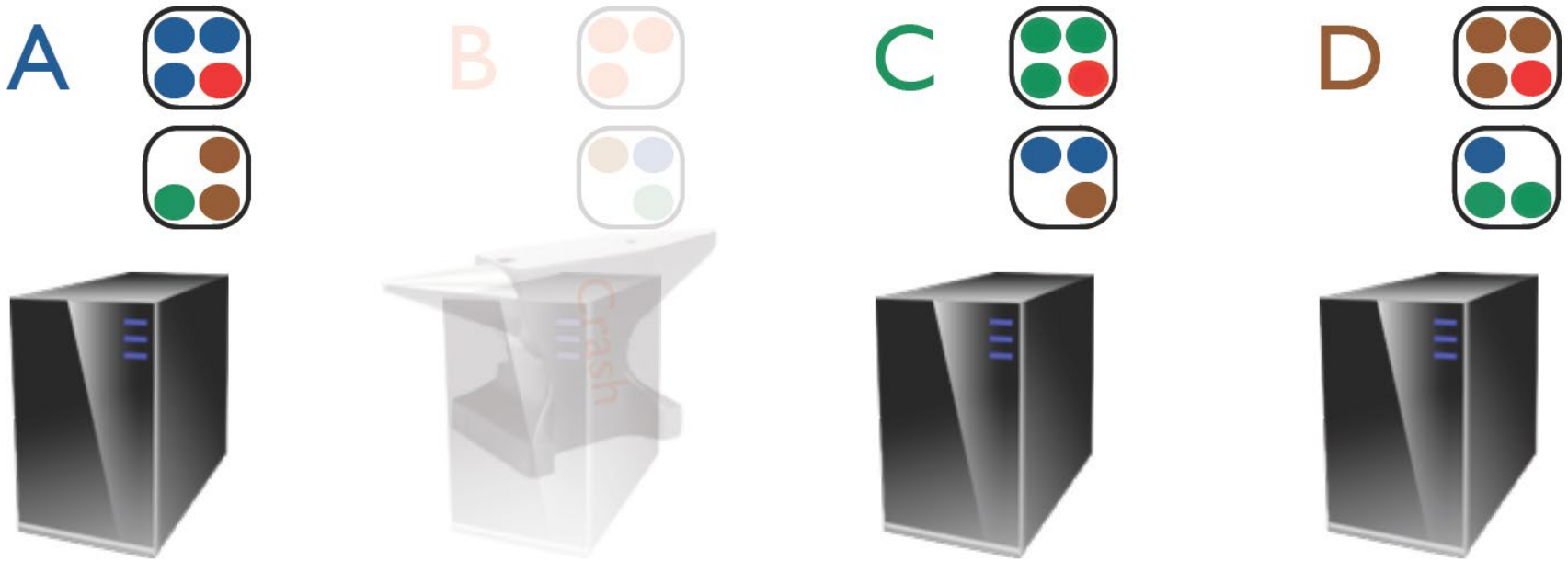
- Knoten A kopiert Partition von D (die auf Knoten B war)



# Wie funktioniert Hazelcast?

Daten wiederherstellen...

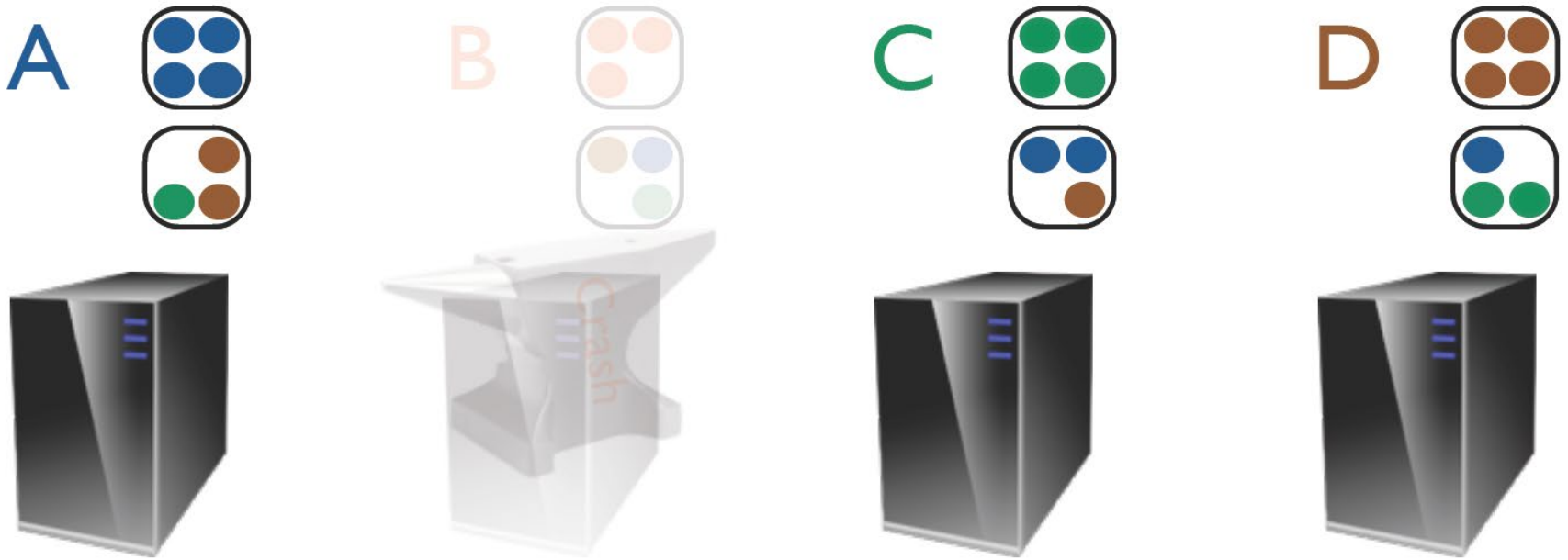
- Backup Partitionen vom ehemaligen Knoten B werden in die eigenen Partitionen übernommen



# Wie funktioniert Hazelcast?

Daten wiederhergestellt – 12 Partitionen

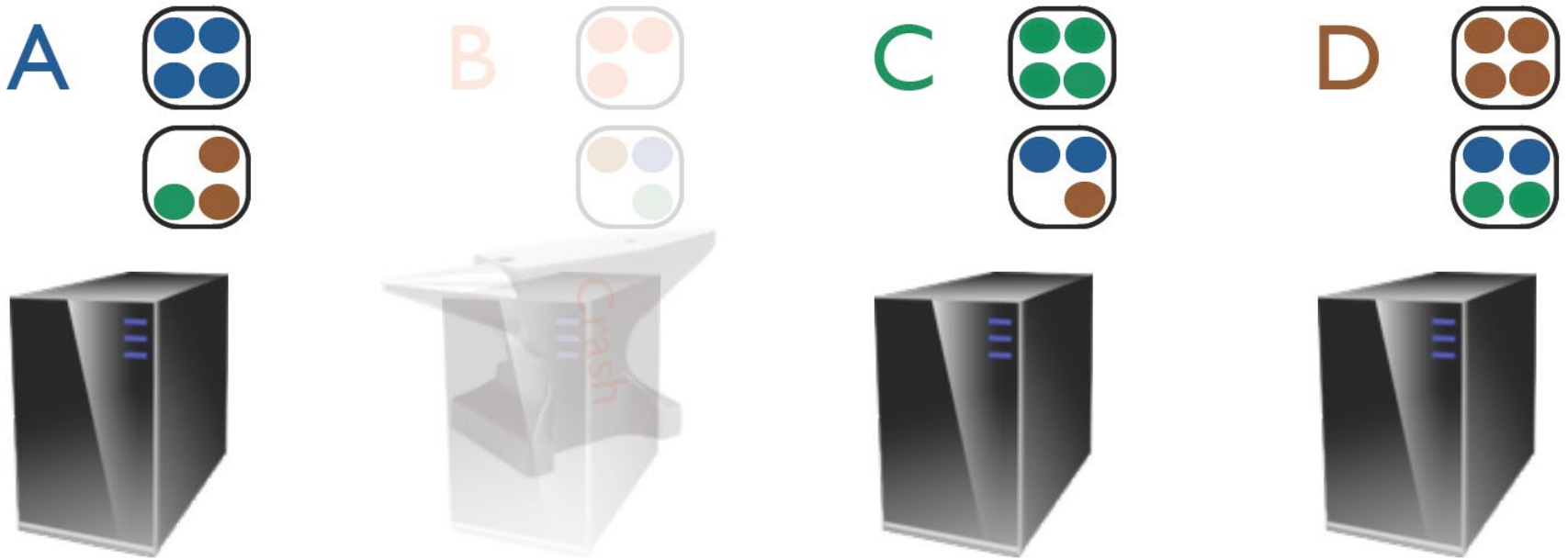
- Die Knoten besitzen wieder alle Partitionen inkl. die vom ehemaligen Knoten B (Besitzwechsel)



# Wie funktioniert Hazelcast?

Backup anlegen von den wiederhergestellten Daten…

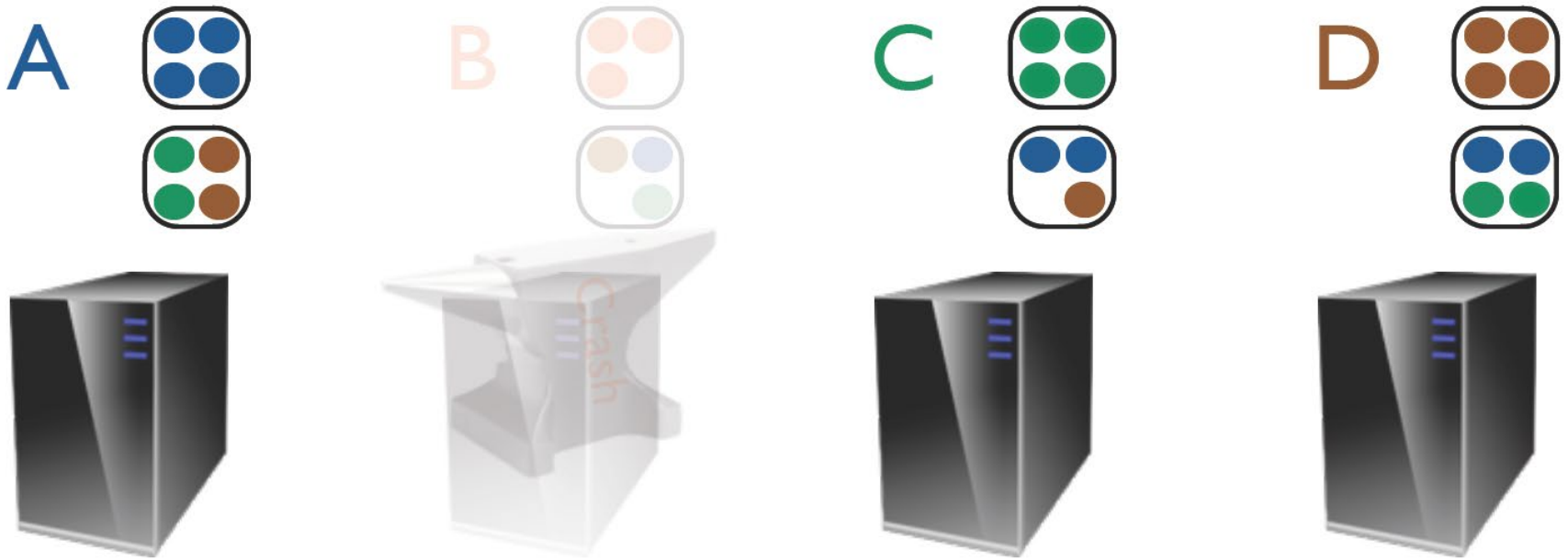
- Knoten D kopiert wiederhergestellte Partition von A



# Wie funktioniert Hazelcast?

Backup anlegen von den wiederhergestellten Daten…

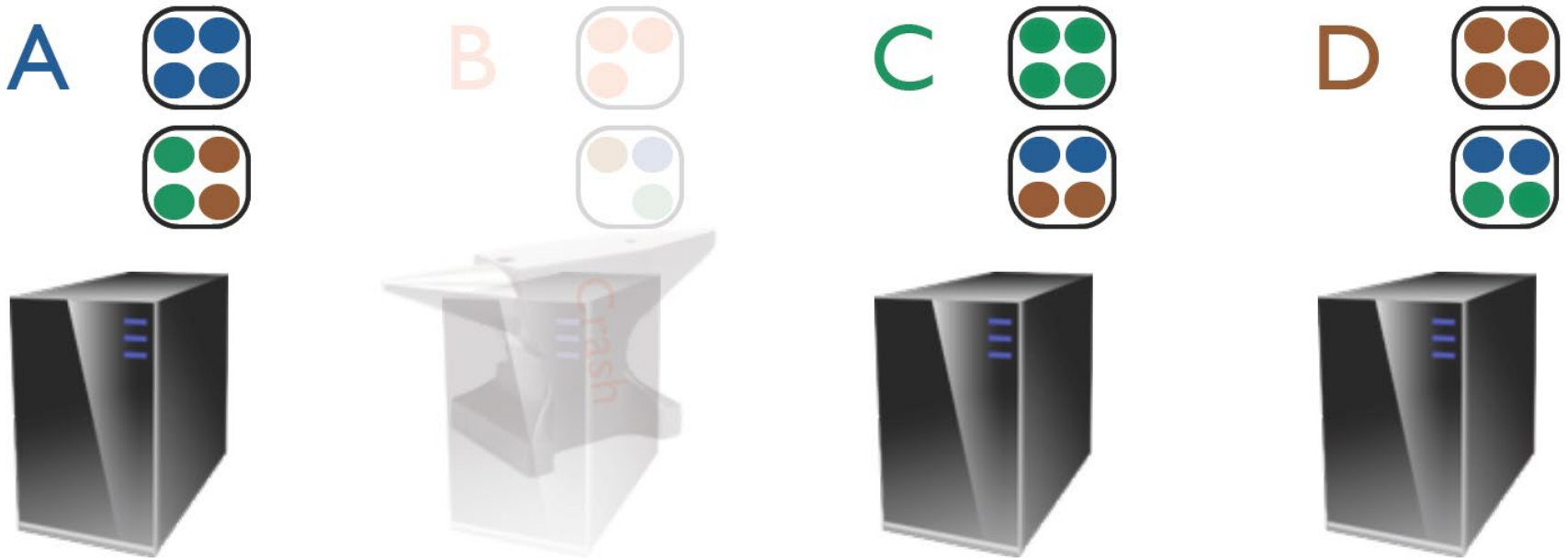
- Knoten A kopiert wiederhergestellte Partition von C



# Wie funktioniert Hazelcast?

Backup anlegen von den wiederhergestellten Daten…

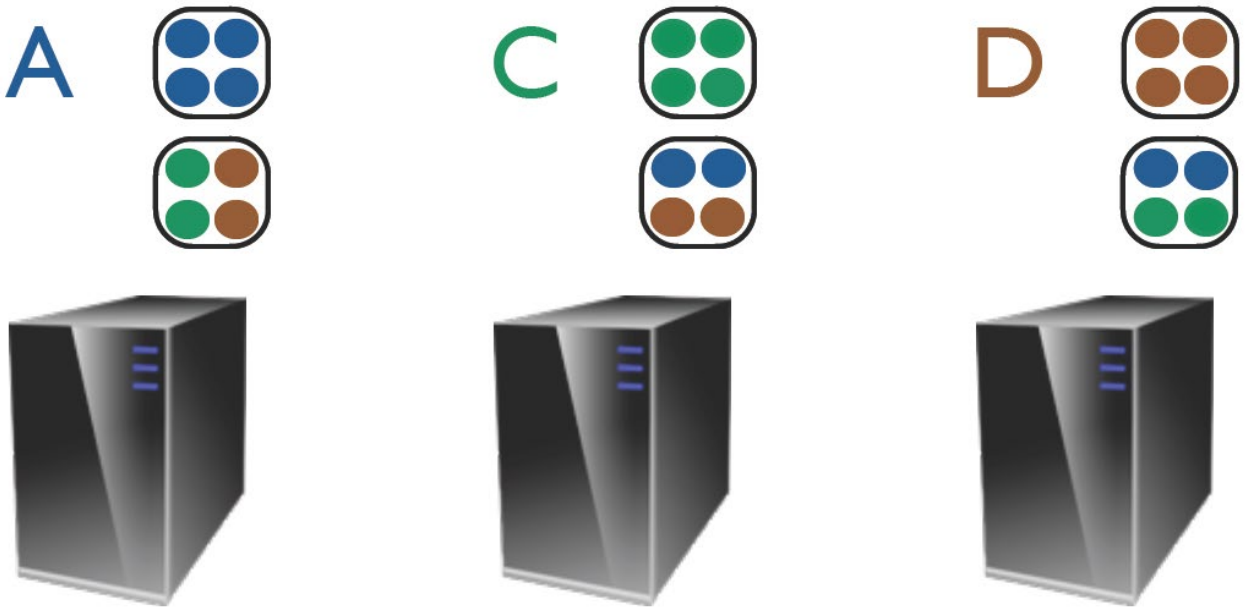
- Knoten C kopiert wiederhergestellte Partition von D



# Wie funktioniert Hazelcast?

Alles wieder vollständig und sicher!

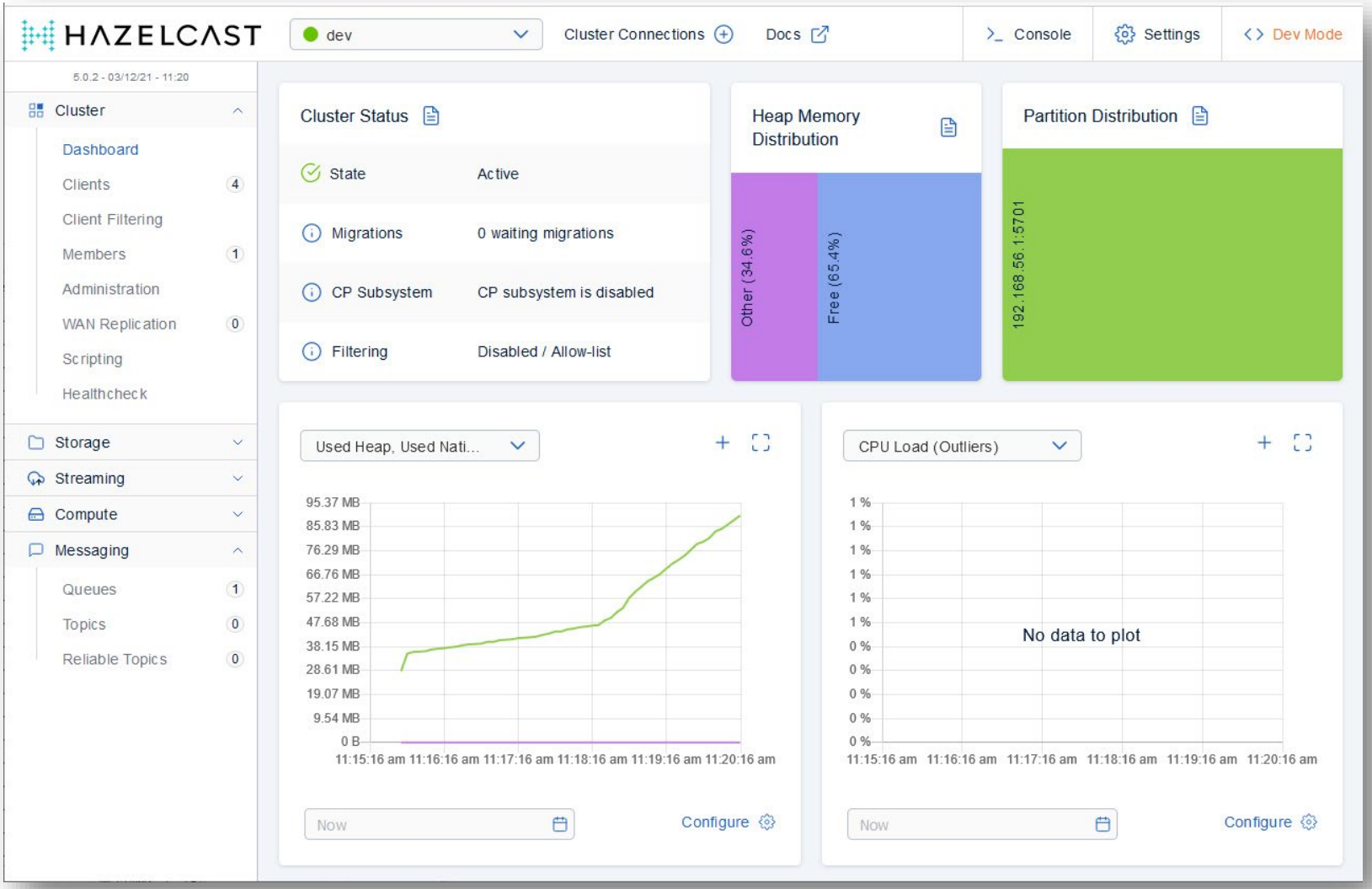
- 12 Partitionen
- 12 Backup Partitionen verteilt auf alle Knoten



# Beispiel: Hazelcast


- In-Memory Data Grid in Java
- Open Source (Apache License 2.0)
  - https://hazelcast.com/open-source-projects/downloads/
- Kommerzielle Versionen & Subscription Plans: Enterprise & HD

# Management Center




https://hazelcast.com/products/management-center/
https://hazelcast.org/imdg/download/

# Programmierung: Aufbau und Abbau einer Connection

– Hazelcast ist Thread safe:

```
Map<String, Employee> map = client.getMap("employees");
List<Customer> list = client.getList("customers");
```

– Viele Instanzen auf der gleichen JVM:

```
HazelcastInstance h1 = Hazelcast.newHazelcastInstance();
HazelcastInstance h2 = Hazelcast.newHazelcastInstance();
//...
```

– Alle zu verteilende Objekte müssen interface DataSerializable implementieren:

```
public class Employee
    implements com.hazelcast.nio.serialization.DataSerializable
```

# Programmierung: Cluster Interface

zeigt Informationen über die Mitglieder des Clusters:

```
MembershipListener listener = new MembershipAdapter();
HazelcastInstance client = Hazelcast.newHazelcastInstance();

Cluster cluster = client.getCluster();
cluster.addMembershipListener(listener);
Member localMember = cluster.getLocalMember();
LOG.info(localMember.getAddress());

Set<Member> members = cluster.getMembers();
Iterator<Member> it = members.iterator();
while (it.hasNext()) {
    Member member = it.next();
    LOG.info(member.getUuid() + " from " + member.getAddress());
}
```

# Programmierung: Distributed Lock

– Falls ein systemweiter Zugriff auf gemeinsame Ressource erfolgen muss, wird dieser mit Locks geschützt:

```
String mylockobject = "MyLock";
FencedLock mylock = client.getCPSubsystem().getLock(mylockobject);
mylock.lock();
try {
    // do something
} finally {
    mylock.unlock();
}
```

– Auch Hazelcast Collections bieten Locks an:

```
IMap<String, Customer> map = client.getMap("customers");
map.lock("1");
try {
    // do something
} finally {
    map.unlock("1");
}
```

# Programmierung: Distributed Map

Collection Map (hier nebenläufig) mit den Möglichkeiten der verteilten Speicherung von Daten:

```
HazelcastInstance client = Hazelcast.newHazelcastInstance();

ConcurrentMap<String, Customer> map = client.getMap("customers");
Customer customer = new Customer("John Doe", 21, false);
map.put("4711", customer);
customer = map.get("4711");
LOG.info(customer);

//ConcurrentMap methods
map.putIfAbsent("4712", new Customer("Chuck Norris", 80));
map.replace("4711", new Customer("Bruce Lee"));
customer = map.get("4711");
LOG.info(customer);
customer = map.get("4712");
LOG.info(customer);
```

# Programmierung: Distributed Queue

Queues können verteilt gespeichert werden und z.B. für verteiltes Ausführen von Tasks genutzt werden.

```
HazelcastInstance client = Hazelcast.newHazelcastInstance();
//Queue operations
BlockingQueue<Task> queue = client.getQueue("tasks");
Boolean done = queue.offer(new Task());
Task task = queue.poll();
LOG.info(task);

//Timed blocking operations
done = queue.offer(new Task(), 500, TimeUnit.MILLISECONDS);
task = queue.poll(5, TimeUnit.SECONDS);
LOG.info(task);

//Indefinitely blocking operations
queue.put(new Task());
task = queue.take();
LOG.info(task);
```

# Programmierung: Distributed Topic

Verteilungsmechanismus für das Veröffentlichen von Nachrichten, an mehrere, registrierte Abonnenten.

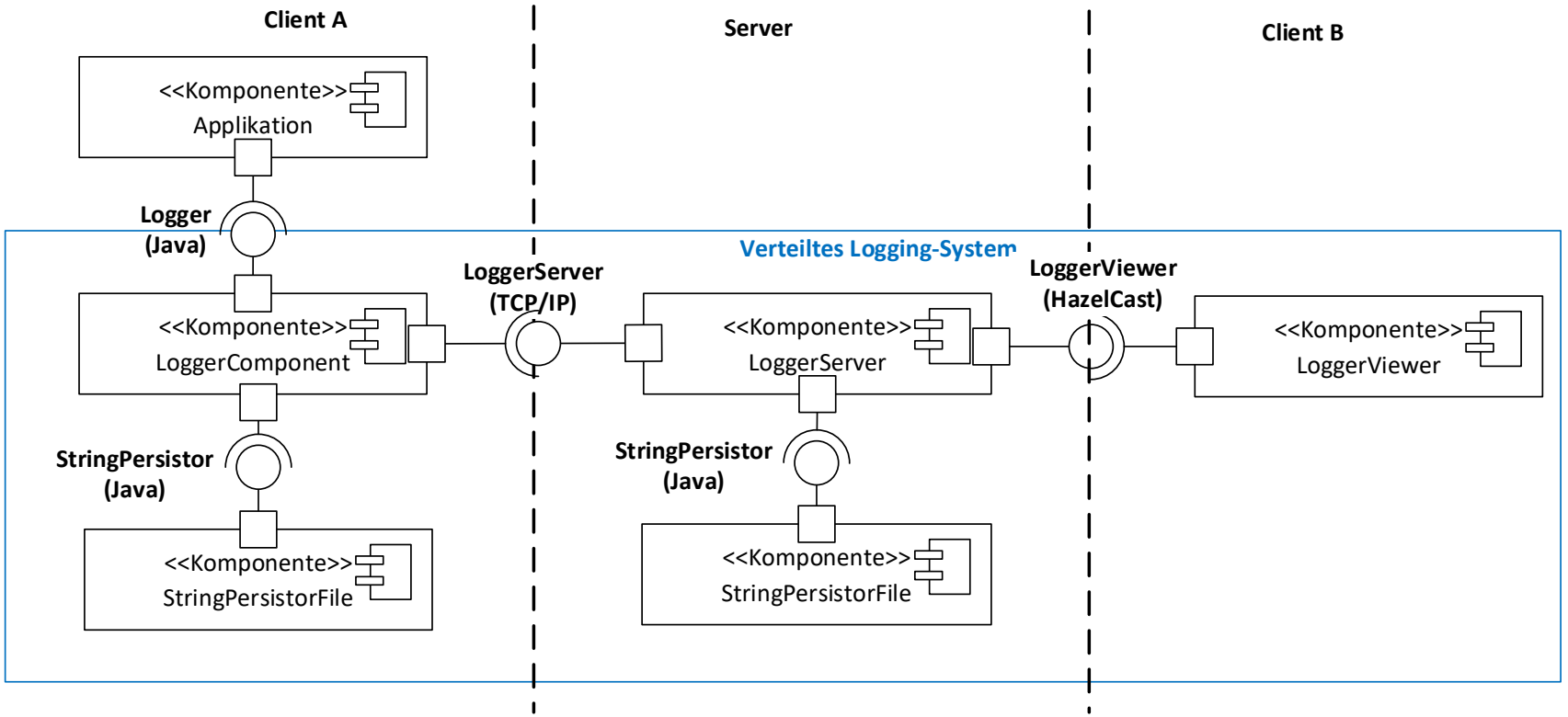
```
public class DistributedTopic implements MessageListener<MyMessage>{
    @Override
    public void onMessage(Message<MyMessage> msg) {
        MyMessage message = msg.getMessageObject();
        LOG.info("Got msg: " + message.getText() +
                 " from " + message.getName());
    }
}

HazelcastInstance client = Hazelcast.newHazelcastInstance();
DistributedTopic distributedTopic = new DistributedTopic();

ITopic<MyMessage> topic = client.getTopic("default");
topic.addMessageListener(distributedTopic);
topic.publish(new MyMessage("my message object", "hello"));
```

# Klassenraumübung: LoggerViewer mittels HazelCast

**Ziel:** Anzeige einer möglichst aktuellen Ansicht der Log-Messages in einem oder mehreren verteilten Clients. Anbindung soll mittels Hazelcast erfolgen.

**Fragestellung:** Welche Datenstrukturen (Map/Queue/Topic) von Hazelcast würden Sie dafür einsetzen und wie?

⇨ Diskutieren Sie in Ihrem Team.



# Zusammenfassung

- Topologie: Separate Verteilung von Daten und Anwendung (Client/Server) oder Zusammenlegen von Daten und Anwendung (Embedded).
- Lastverteilung dient der Skalierung einer verteilten Applikation.
- Skalierung erschwert durch temporären oder persistenten Zustand.
- Reverse-Proxy sind eine Middleware, welche typischerweise die Lastverteilung mittels verschiedenen Methoden ermöglicht.
- In-Memory Data-Grid speichern Daten in Partitionen im Hauptspeicher.
  - Beispiel einer Embedded-Topologie.
  - Daten sind gleichmässig über die Nodes des Systems verteilt.
  - Zur Ausfallsicherheit sind die Daten redundant gespeichert.
- In-Memory-Data-Grids bieten oft vielfältige Möglichkeiten zur verteilten Datenspeicherung, welche sich zur Kommunikation zwischen Systemen eignet.

# Fragen?