HSLU Hochschule Luzern

Verteilte Systeme und Komponenten

Konsistenz und Replikation

Martin Bättig

Letzte Aktualisierung: 17. November 2022

FH Zentralschweiz

Inhalt

- CAP-Theorem
- Konsistenz und Konsistenzgarantien
- Replikation
- Ausfallsicherheit mittels Replikation

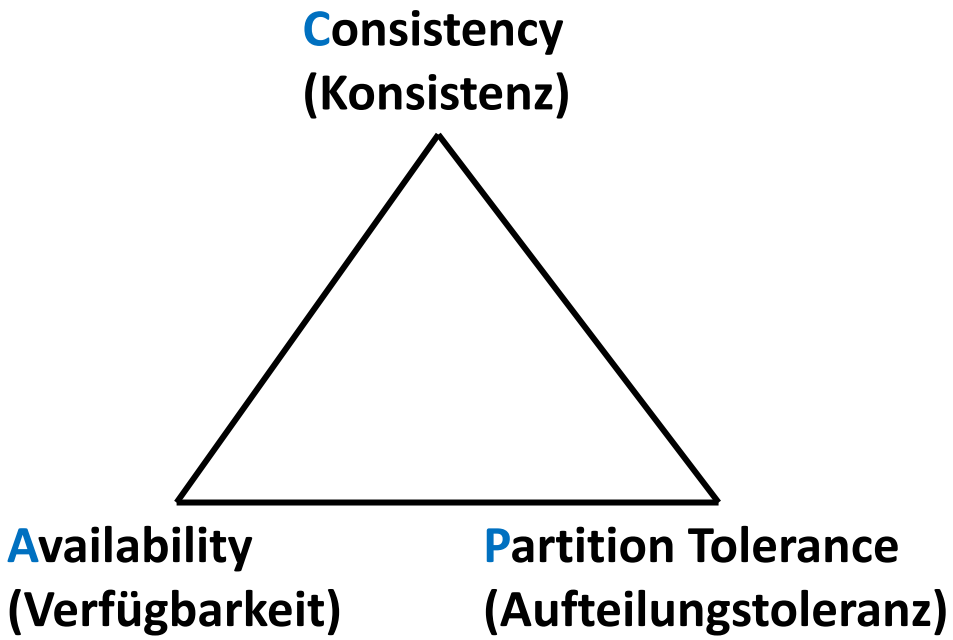
Lernziele

- Sie kennen das CAP-Theorem (Consistency, Availability, Partition Tolerance) und können verteilte Systeme anhand dieses Protokolls einordnen.
- Sie kennen den Begriff der Konsistenz sowie die Konsistenzgarantien «sequential consistency» und «eventual consistency» und können einschätzen, bei welchen Arten von verteilten Systemen diese Konsistenzgarantien eingesetzt werden.
- Sie kennen den Prozess der Replikation und verschiedene Möglichkeiten ein Replika zu erzeugen.
- Sie können Replikation in Zusammenhang mit den Konsistenzanforderungen setzen.
- Sie kennen die Mechanismen Heartbeat und Leader-Election und wie Sie damit Replikas zum Zweck der Ausfallsicherheit eines verteilten Systems einsetzen können.

CAP-Theorem

CAP-Theorem

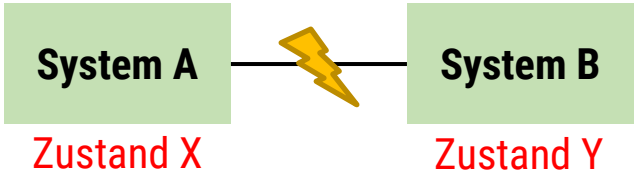
Brewer's Theorem (2002) besagt, dass ein verteiltes System nicht mehr als zwei der folgenden Eigenschaften garantieren kann:



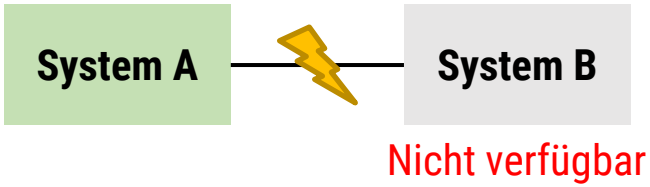
- **Konsistenz:** Alle beteiligen Systeme haben identische und aktuelle Daten.
- **Verfügbarkeit:** Daten sind für alle Lese-/Schreiboperationen sofort bereit.
- **Aufteilungstoleranz:** System kann trotz Aufteilung in zwei oder mehr bzgl. Kommunikation getrennte Partitionen weiter operieren.

CAP-Theorem: Intuition

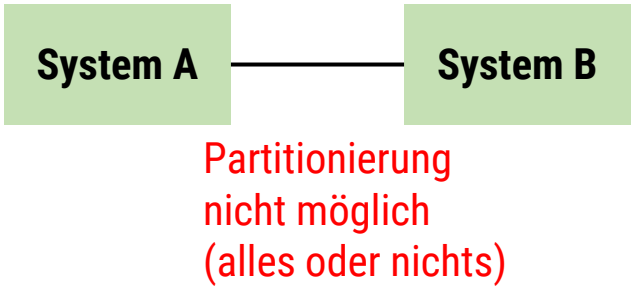
- **AP-Systeme** operieren trotz ausgefallener Kommunikation weiter.
  Reduktion der Konsistenz.
  (Beispiele: Domain-Name-System, Consumer-Filesynchronisation)



- **CP-Systeme** operieren trotz ausgefallener Kommunikation weiter, aber nur soweit die Konsistenz gewährleistet ist.
  Reduktion der Verfügbarkeit.
  (Beispiele: Email, Bankomat)



- **CA-Systeme** bieten sowohl Konsistenz als auch Verfügbarkeit dürfen darum nicht partitioniert werden (*).
  Reduktion der Partitionstoleranz.
  (Beispiel: Typische Webapplikation).



(*) **Relevanz sinkend:** https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed/

Konsistenz und Konsistenzgarantien

Konsistenz

- Sobald ein System A und ein System B über die gleichen Daten verfügen (z.B. wenn B ein Replikat von A ist), stellt sich die Frage bzgl. deren Konsistenz.
- Liefern A und B bei identischen Anfragen die identische Antwort sind die Daten der beiden Systeme konsistent.
- Systeme können unterschiedliche Garantien bzgl. der Konsistenz der Daten geben.

Konsistenzgarantien

- Anforderungen an die verteilte Applikation bestimmen die notwendige Art der Konsistenz.
- Typische Konsistenzgarantien sind:
  - **Sequential Consistency:** Die Sequenz der Operationen eines Systems S ist für alle Systeme die gleiche und passt in eine Gesamtordnung.
  - **Eventual Consistency:** Modifikationen werden erst mit einer gewissen Verzögerung sichtbar.

Sequential Consistency

«Sequential consistency» gibt folgende Konsistenzgarantie:

Das Resultat einer verteilten Ausführung ist das gleiche wie:

- wenn Lese- und Schreiboperationen aller beteiligen Systeme einer globalen sequentiellen Abfolge zugeordnet werden können.
- Und in dieser Abfolge für jedes System die Operationen in der gleichen Reihenfolge, in welcher es die Operationen getätigt hat, vorkommen.
- Die Zeit spielt dabei keine Rolle!

Typische Anforderung einer verteilten Eingabe und Verarbeitungs-Applikation.

Vorteil: Unabhängige Teiloperationen blockieren das Gesamtsystem nicht.

Beispiel: Sequentielle Konsistenz bei Konten

Annahme: Applikation, in welcher wir in ein Konto lesen oder schreiben.

Notation:

K ⇨ B	lese von Konto K und erhalte Betrag B.
B ⇨ K	schreibe Betrag B in Konto K.

Anforderung an sequentielle Konsistenz erfüllt:

System A:	X ⇨ 10	Y ⇨ 20	15 ⇨ X	15 ⇨ Y
System B:	20 ⇨ Y	X ⇨ 10	Y ⇨ 20	20 ⇨ Z

Anforderung an sequentielle Konsistenz nicht erfüllt:

System A:	X ⇨ 10	Y ⇨ 20	15 ⇨ X	15 ⇨ Y
System B:	X ⇨ 10	Y ⇨ 25	15 ⇨ X	20 ⇨ Y

Eventual Consistency und Monotonic Reads

Beschreibung (informell):

- Das Gesamtsystem konvergiert gegen die aktuellen Daten. Die Konsistenzanforderung «Eventual Consistency» erlaubt, dass ein System zu einem Zeitpunkt T noch mit veralteten Daten arbeitet.
- Typischerweise in Verbindung mit Monotonic Reads: Aufeinander folgende Leseoperationen auf ein Objekt O sehen entweder den gleichen oder einen aktuelleren Wert als den zuletzt gelesen.

Nutzen:

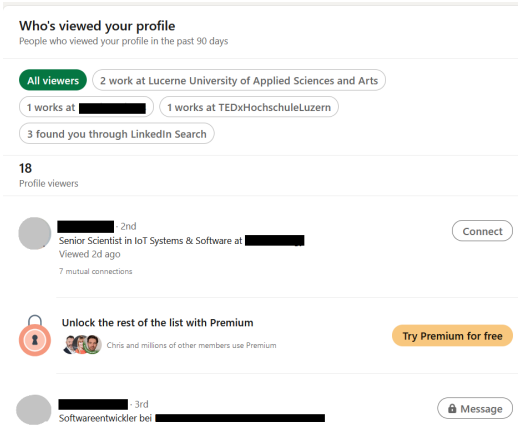
- Zeitliche Entkoppelung, Verfügbarkeit, tolerieren von Netzpartitionierung.

Einsatzszenario:

- Falls Daten nicht immer aktuell sein müssen UND
  - falls nur eine Instanz die Hoheit über Modifikationen besitzt

  ODER
  - nur Schreiboperationen getätigt werden.

Fallbeispiel: «Zuletzt besucht»



Besucher eines Profils P werden Liste L gespeichert.

Bei Betrachten von Profil P, sieht der Besitzer Liste L.

Notation:

$P \Rightarrow L_t$	erhalte Liste der letzten Besucher von Profil P zum Zeitpunkt t.
$L_t \Rightarrow P$	füge Besucher B zur Liste von Profil P hinzu.

Anforderung an Eventual Consistency mit Monotonic Reads erfüllt:

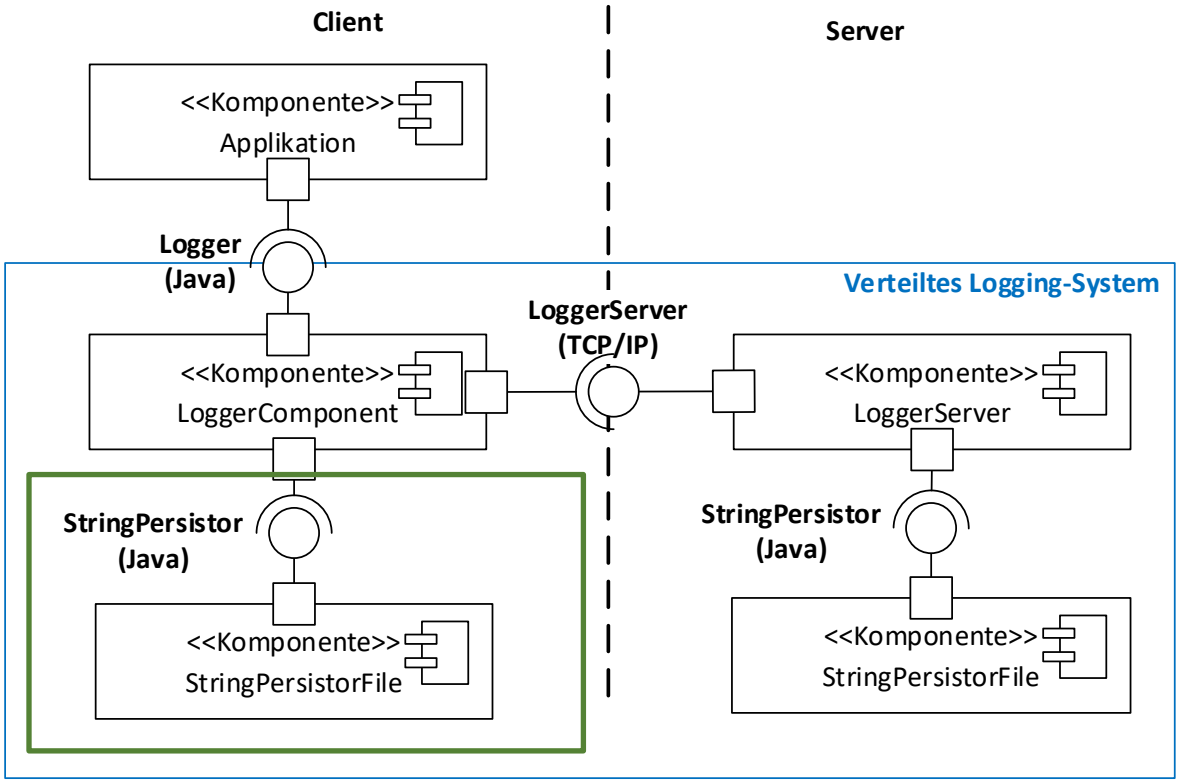
System A:	$L_1 \Rightarrow P$	$L_2 \Rightarrow P$	$L_3 \Rightarrow P$	
System B:	$P \Rightarrow L_1$	$P \Rightarrow L_3$	$P \Rightarrow L_3$	$P \Rightarrow L_3$

Anforderung an Eventual Consistency mit Monotonic Reads nicht erfüllt:

System A:	$L_1 \Rightarrow P$	$L_2 \Rightarrow P$	$L_3 \Rightarrow P$
System B:	$P \Rightarrow L_3$	$P \Rightarrow L_1$	

Übung: Konsistenzgarantien des verteilten Logger-Systems

- Welche Konsistenzgarantien wünschen Sie sich für das verteilte Logging-System?
- Mit welcher Begründung?



Replikation

Replika und Replikation

Replika:

- Eine Kopie eines Systems.
- Diese muss nicht Bit für Bit identisch sein, sollte aber die gleichen Informationen enthalten
  Beispiel: Abbild eines Dateisystems erstellen vs. kopieren Datei-für-Datei.

Replikation:

- Der Prozess aus einem Original (Primary) eine Kopie (Replikat) herzustellen.
- Hauptanwendungen der Replikation bei verteilten Systemen:
  - **Ausfallsicherheit:** Fällt Primary A aus, kann ein Replikat B übernehmen.
  - **Performance:** Verteilung von Anfragen über mehrere Systeme hinweg.

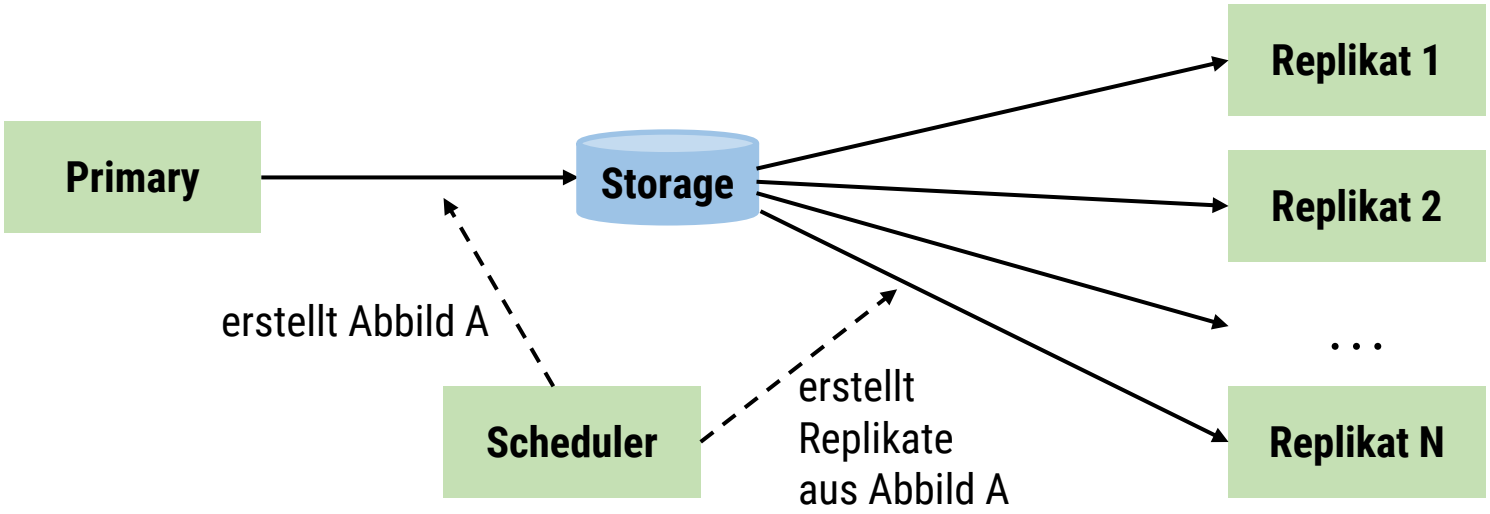
Fragestellungen im Zusammenhang mit Replikation

- Wie konsistent müssen Primary und Replikate zu einem bestimmten Zeitpunkt sein?
- Wie kann ein Ausfall eines Systems erkannt werden?
- Wie kann der Ausfall eines Systems toleriert werden?
- Wo müssen Primary und Replikate platziert sein? (=> Thema des Inputs «Skalierung und Verteilung»)
- Wie wird die Last verteilt? (=> Thema des Inputs «Skalierung und Verteilung»)

«Klassische» Replikation

Ein Scheduler führt in regelmässigen Intervallen (z.B. alle 6h) folgende Anweisungen aus:

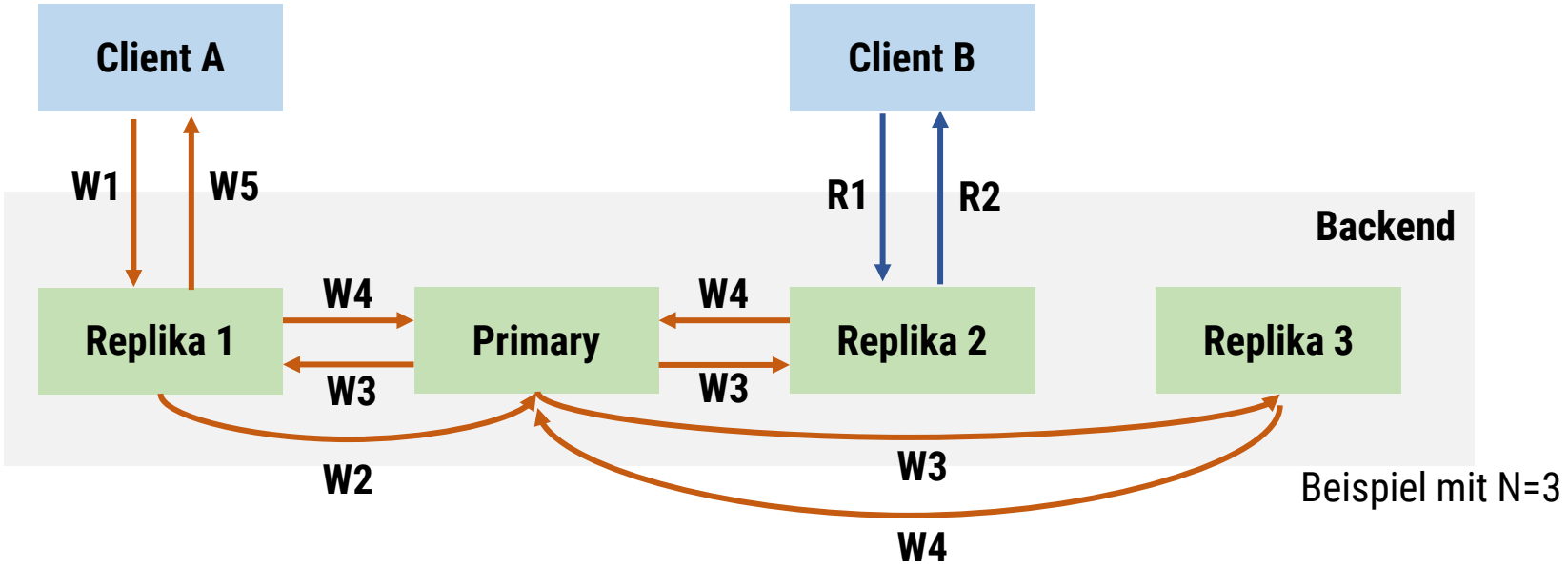
1. Erstellung eines Abbilds A des Primarys.
2. Einspielung von Abbild A in alle Systeme, welche als Replikat dienen sollen.



– Wie konsistent sind diese Replika?
– Was gilt es bei dem Prozess zu beachten?

Primary-Backup-Protokoll für «On-the-Fly» Replikas

- 1 Primary: Behandelt alle Schreibzugriffe.
- N Replikate: Beantwortet Lesezugriffe, leitet Schreibzugriffe an Primary weiter.



W1: Schreibanfrage

W2: Weiterleitung an Primary

W3: Update von Primary an Replikat

W4: Replikat bestätigt Update

W5: Bestätige Schreibanfrage

R1: Leseanfrage

R2: Beantwortet Leseanfrage

Eigenschaften des Primary-Backup-Protokolls

Standard-Implementation: Antwort auf Schreibanfrage kehrt erst zurück, sobald Replikas die Daten geschrieben haben -> blockierend.

- Push-Prinzip.
- Sequentielle Konsistenz.
- Aber Ausfallsicherheit?
- Auswirkung auf Antwortszeit?

Fallstrick: Filesystem-Cache

- Betriebssysteme haben i.d.R. einen Filesystem-Cache (Zwischenspeicher).
- Änderungen werden zuerst in den Cache und erst nach gewisser Zeit auf Disk geschrieben.
- Für eine sequentielle Konsistenz (z.B. bei Datenbanken) muss Schreiben auf Disk erzwungen werden.

Beispiel: java.nio.channels.FileChannel

force

```
public abstract void force(boolean metaData)
                   throws IOException
```

Forces any updates to this channel's file to be written to the storage device that contains it.

If this channel's file resides on a local storage device then when this method returns it is guaranteed that all changes made to the file since this channel was created, or since this method was last invoked, will have been written to that device. This is useful for ensuring that critical information is not lost in the event of a system crash.

Viele Variationen des Primary-Backup-Protokolls

Implementiert von vielen Produkten (z.B. DBs wie Oracle/MySQL/etc.), aber nicht in der ursprünglichen Form.

- **Nicht blockierender Schreibzugriff:**
  - Ggf. Datenverlust (Daten nicht auf Replikat) nach Crash von Primary oder Primary muss nach Crash wiederherstellt werden (Downtime).
- **Push- vs. Pull-Prinzip:**
  - In festgelegtem Intervall: Keine sequentielle Konsistenz, aber je nach Daten und Anwendungszweck immer noch «Eventual Consistent».
- **Einzelner vs. verteilte Primarys:**
  - Jedes teilnehmende System kann für gewisse Objekte Primary und für andere Objekte Replikat sein.
  - Komplexer, aber schneller durch verteilte Schreibzugriffe.

Push-Replikation: Replikationsqueue und Verbindungsaufbau

```
public class Replication implements Runnable {

    private final String replicationAddress;
    private final Queue<Message> replicationQueue
        = new ConcurrentLinkedQueue<>();



    public Replication(String replicationAddress) {
        this.replicationAddress = replicationAddress;
    }

    @Override
    public void run() {
        try (ZContext context = new ZContext()) {
            ZMQ.Socket socket = context.createSocket(SocketType.REQ);
            socket.connect(replicationAddress);
            processReplicationQueue(socket);
        } catch (Exception ex) {
            LOG.error(ex.getMessage(), ex);
        }
    }
```

Queue für alle zu synchronisierenden Aktivitäten.

Verbindung zum Replikat aufbauen und replizieren.

Push-Replikation: Messages hinzufügen und Verarbeitung

```
public void addMessage(Message message) {
    replicationQueue.add(message);
    notifyWaiters();
}

public void waitForSynchronization() {
    waitForCondition(true);
}

private void processReplicationQueue(ZMQ.Socket socket) {
    while(true) {
        waitForCondition(false);
        Message message = replicationQueue.peek();
        socket.send(message.toString());
        socket.recvStr();
        replicationQueue.remove();
        notifyWaiters();
    }
}

private synchronized void waitForCondition(boolean isEmpty) {
    while (replicationQueue.isEmpty() != isEmpty) this.wait();
}
```

Message zur Replikationsqueue hinzufügen

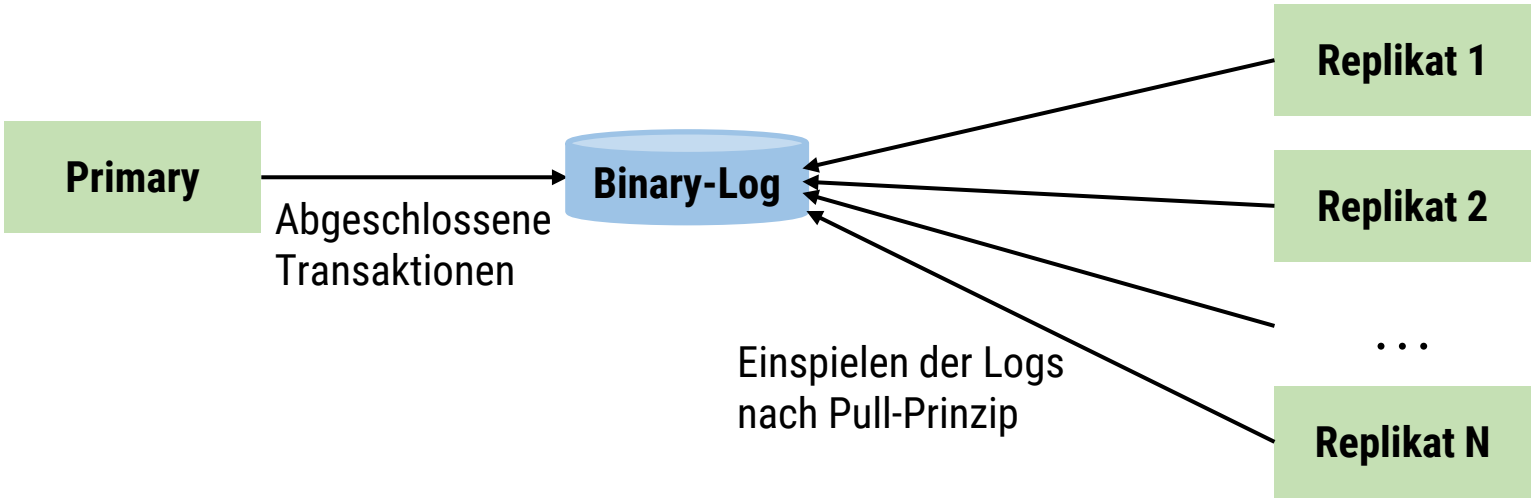
Warten bis Synchronisation abgeschlossen ist.

Verarbeiten: Synchrones Senden von Messages zum Replikat.

Fallstudie: Standard-Replikation bei Datenbank «MariaDb»

Grober Ablauf:

- Abgeschlossene Transaktion werden in ein Binary-Log geschrieben.
- Replikats lesen neue Informationen im Binary-Log nach dem Pull-Prinzip und applizieren es lokal.
- Konsistenz: Replikas verfügen nur mit Verzögerung über die gleichen Daten.



Klassenraumübung: Eigene Versuche mit Replikation

Übung mittels Online-Programmierumgebung:

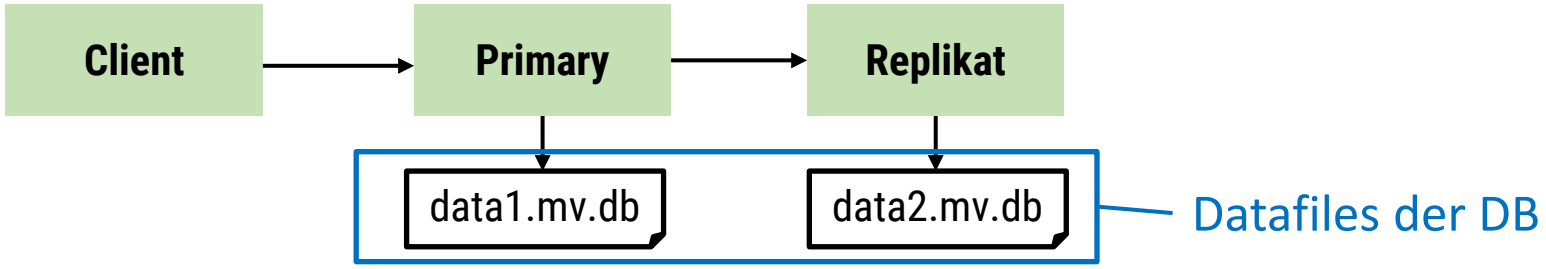
- https://replit.com/@mbaettig/Replication
  - **Kein HSLU-Dienst (benötigt separates Konto), erstellen Sie einen Fork.**

Aufgabe: Machen Sie eigene Versuche mit dieser *Teilimplementation* des Primary-Backup-Protokolls:

Beim Start mittels folgenden Anweisungen:

```
java AccountServer ./data1 tcp://localhost:5555 tcp://localhost:5556 primary
java AccountServer ./data2 tcp://localhost:5556 tcp://localhost:5555 replica
java AccountClient tcp://localhost:5555
```

erhält man folgendes Setup:



Klassenraumübung: Eigene Versuche mit Replikation

Machen Sie beispielweise folgendes:

1. Starten Sie den Primary, Replika und Client jeweils in einer eigenen Shell.
2. Erstellen Sie zwei Konten und Übertragen Sie einen Betrag.

```
create acc1 1000            erstellt Konto acc1 mit 1000 CHF
create acc2 500             erstellt Konto acc2 mit 500 CHF
transfer acc1 acc2 300      überweist 300 CHF von acc1 zu acc2
balance acc1                zeigt Kontostand von acc1 an
```

3. Stoppen Sie den Primary (Simulation eines Crashes).
4. Stoppen Sie den Client und starten Sie ihn mit Verbindung auf das Replika.
5. Machen eine weitere Überweisung.

Machen sie sich folgende Überlegungen:

- Wie konsistent ist das Replika? Wie könnte man das ändern?
- Wie könnten Sie den Primary wieder konsistent erhalten? Gibt es Varianten?
- Könnten Sie mehrere Replikas mit diesem System verwalten? Performance?

Ausfallsicherheit mittels Replikation

Replikate für Ausfallsicherheit

Fragestellung: Wie erfolgt automatischer Wechsel auf Replikat, sobald Primary ausfällt?

Teilfragen:

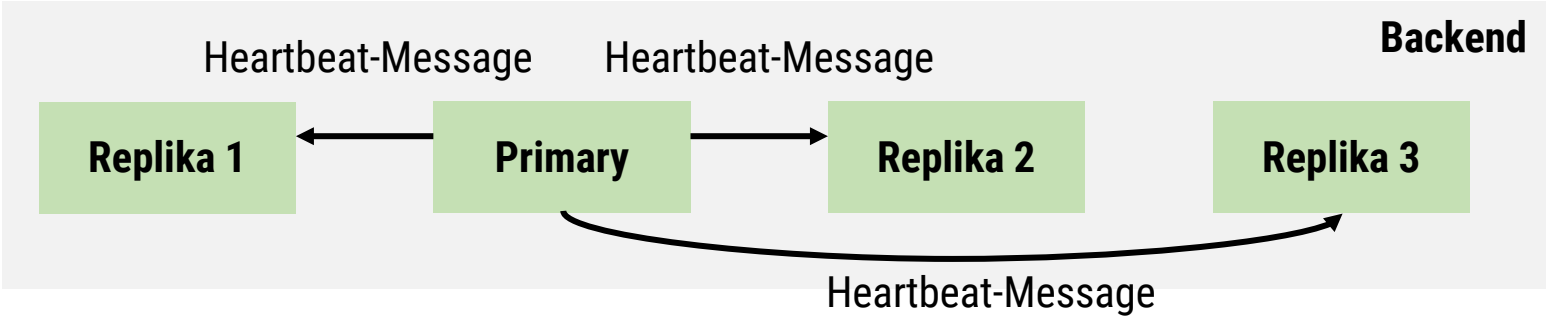
- Wie erkennen Replikate zuverlässig, dass der Primary ausgefallen ist?
  ⇨ Heartbeat-Protokolle
- Wie ernennen die Replikate neuen Primary, falls mehrere Replikate existieren?
  ⇨ Leader-Election-Protokolle
- Wie wissen Clients auf welches System sie verbinden müssen?
  ⇨ Namesdienste / IP-Routing

Heartbeat-Protokoll

Ziel: Erkennen eines Ausfalls des Primarys.

Vorgehen:

- *1 Primary* sendet Heartbeat-Message in festem Intervall (z.B. 100ms, abhängig von der Roundtrip-Time zwischen den Systemen).
- *N Replika* überprüfen, ob diese mindestens ein Heartbeat-Signal innerhalb eines Timeout-Intervalls erhalten haben (z.B. 1000ms). Ist dies nicht der Fall, wird der Server als Down betrachtet.



Beispiel mit N=3

Beispiel: Einfacher Heartbeat Sender (Auszug)

```
private static final long HEARTBEAT_INTERVAL = 100; // [ms]

private final ScheduledThreadPoolExecutor executor
    = new ScheduledThreadPoolExecutor(1);

private ZContext zContext;
private ZMQ.Socket socket;

private final Runnable action = new Runnable() {
    public void run() {
        socket.send(new byte[0]);
    }
};

public void start() {
    zContext = new ZContext();
    socket = zContext.createSocket(SocketType.PUB);
    socket.bind("tcp://localhost:7777");
    executor.scheduleWithFixedDelay(action, HEARTBEAT_INTERVAL,
        HEARTBEAT_INTERVAL, TimeUnit.MILLISECONDS);
}
```

Sende Heartbeat
(Message ohne Inhalt)

Sende Heartbeat
über separaten Kanal

Beispiel: Einfacher Heartbeat Detector (Auszug)

```
private static final long HEARTBEAT_DETECTION_INTERVAL = 1000; // [ms]
private ZContext zContext;
private ZMQ.Socket socket;
private final Runnable action = new Runnable() {
    public void run() {
        while(running) {
            if (socket.recv() == null) {
                running = false;
                noResponseHandler.run();
            }
        }
    }
};
public void start() {
    zContext = new ZContext();
    socket = zContext.createSocket(SocketType.SUB);
    socket.subscribe(new byte[0]);
    socket.connect("tcp://localhost:7777");
    socket.setReceiveTimeOut(HEARTBEAT_DETECTION_INTERVAL);
    Thread thread = new Thread(action);
    running = true;
    thread.start();
}
```

Ist das Timeout eingetreten wird die Behandlungsroutine gestartet.

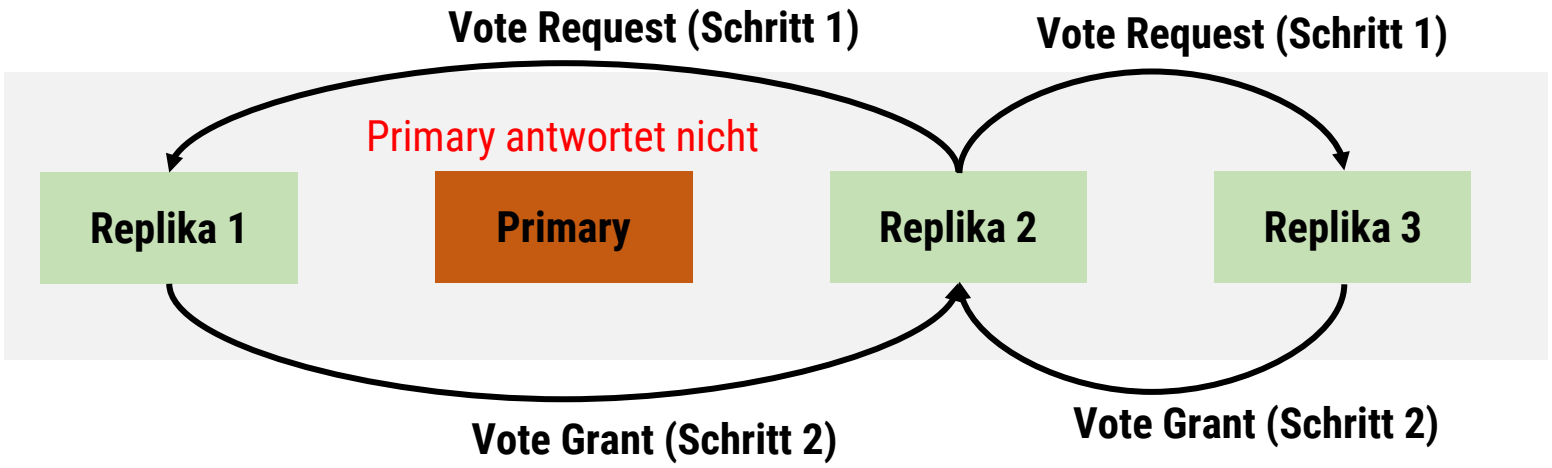
Setzte Timeout auf maximale Zeit ohne Heatbeat

Leader-Election mittels Quorum

Ziel: Ernennen eines neuen Leaders mittels Quorum (Mehrheitsentscheid)

Vorgehen:

- *1 (oder mehr) Replika R* erkennen Ausfall des Primary.
- *Replika R* sendet *Vote Request* an alle anderen Replika.
- Die anderen Replika senden *Vote Grant,* falls diese mit Wechsel einverstanden sind (keine anderen Requests; erkennen Primary als Down).
- *Replika R* wird neues Primary, falls es **ceil(N / 2)** *Vote Grants* erhält.



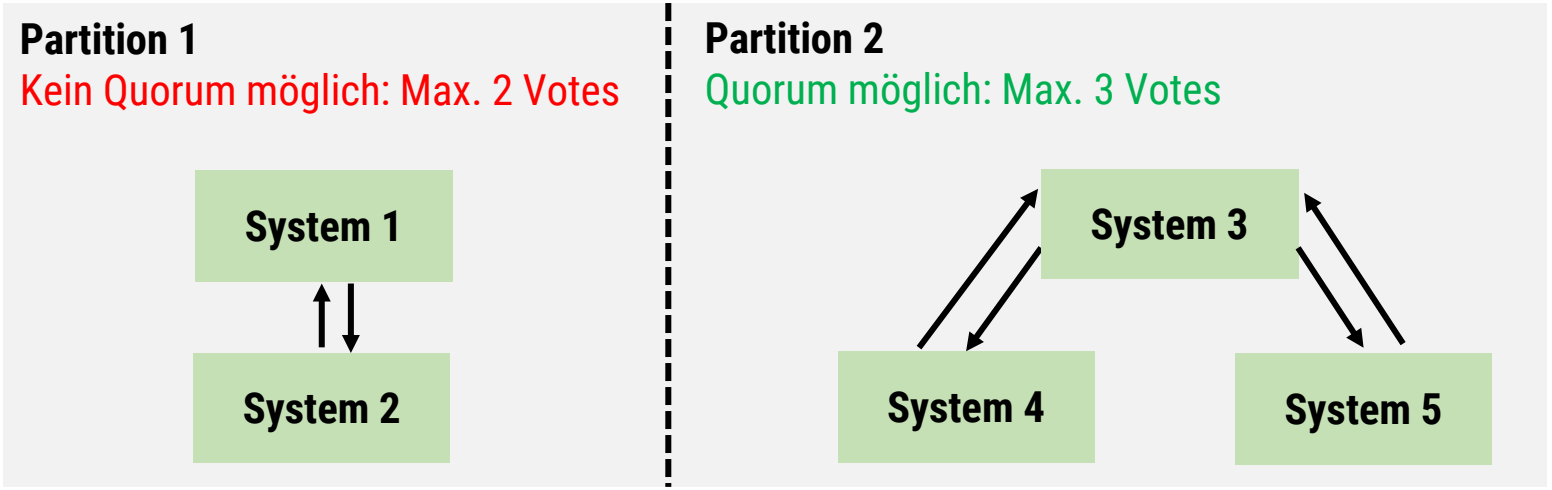
Beispiel mit N=3

Notwendigkeit eines Quorums

- Bei Netzwerkpartitionierung darf nur eine Partition weiter operieren.
- Quorum ab N=3 Systemen möglich, wobei N ist sinnvollerweise ungerade ist.

Beispiel: Verteiltes System mit 5 Systemen wird durch Netzwerkausfall in 2 Partitionen unterteilt.

- Quorum: ceil(N / 2) = 3
- Zwischen Partitionen ist keine Kommunikation mehr möglich.

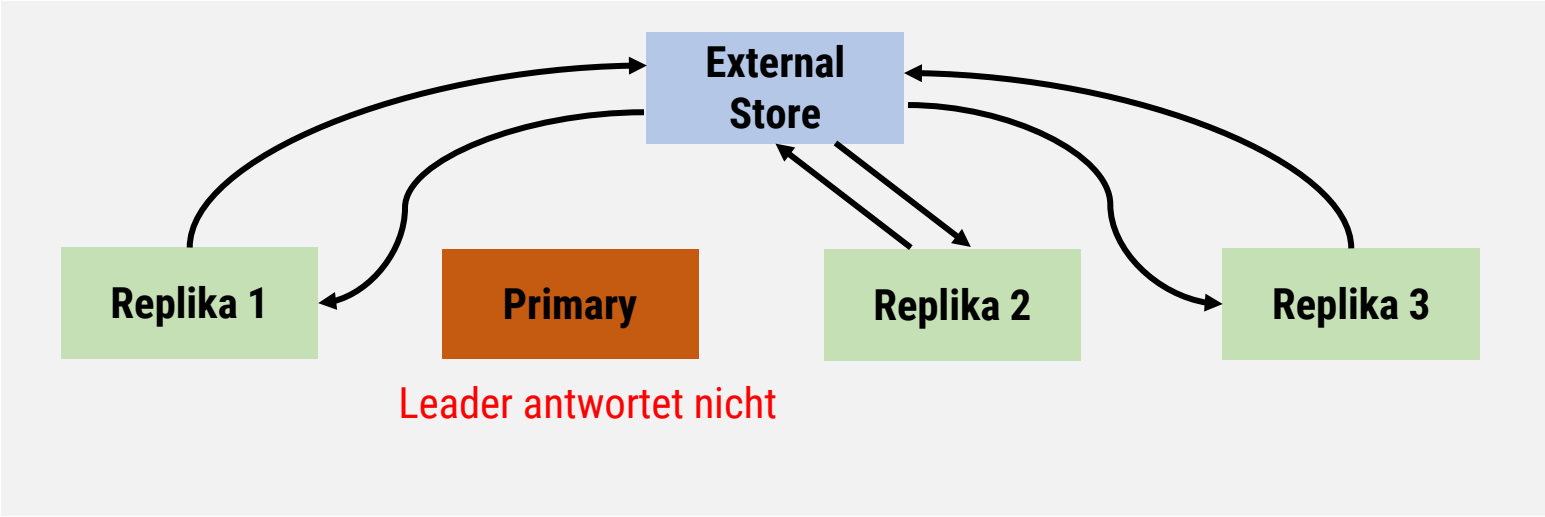


Leader-Election mittels externem Store

Idee: Kleiner aber ausfallsicher zentraler Store implementiert Quorum-Algorithmus. Dieser Store wird von grösseren System verwendet.

– **Beispiele:** ZooKeeper (generischer Store in Java) oder etcd (Kubernetes).

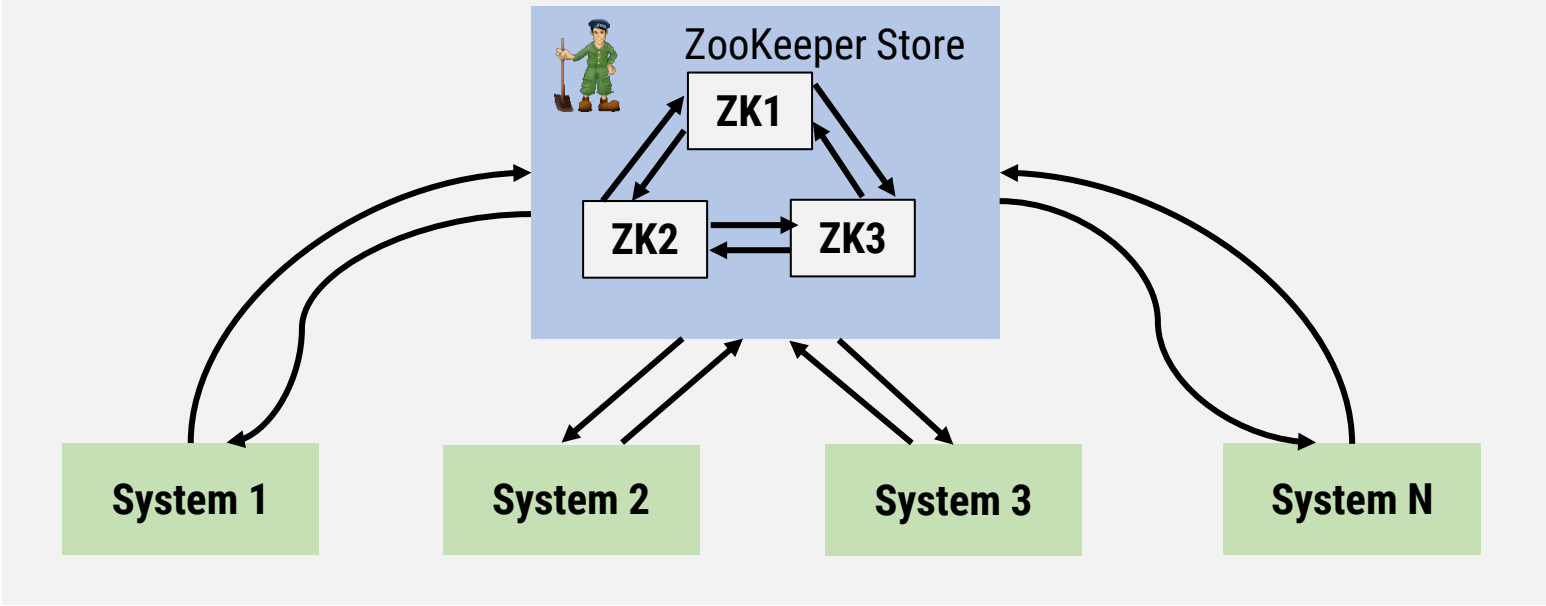
Ablauf: *N Replika* senden Election-Request an zentralen Store, einer der Replika wird als Primary eingetragen.



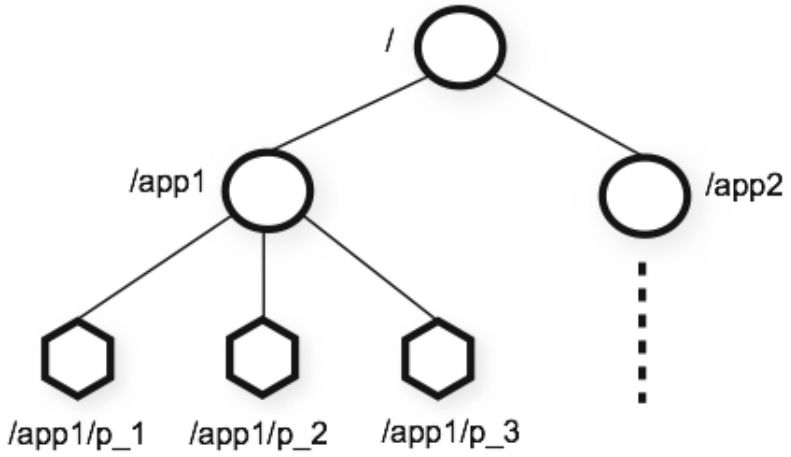
Beispiel mit N=3

Beispiel: ZooKeeper

- Zentraler Informationsdienst in Java.
- Ausfallssicher durch Konsensus-Protokoll.
- Bietet verteilten Systemen folgende Funktionalität:
  - Namensdienste (Auffinden von Systemen).
  - Verteilte Synchronisation (z.B. systemübergreifende Locks).
  - Verwaltung von Gruppenzugehörigkeit (z.B. für Replikas).

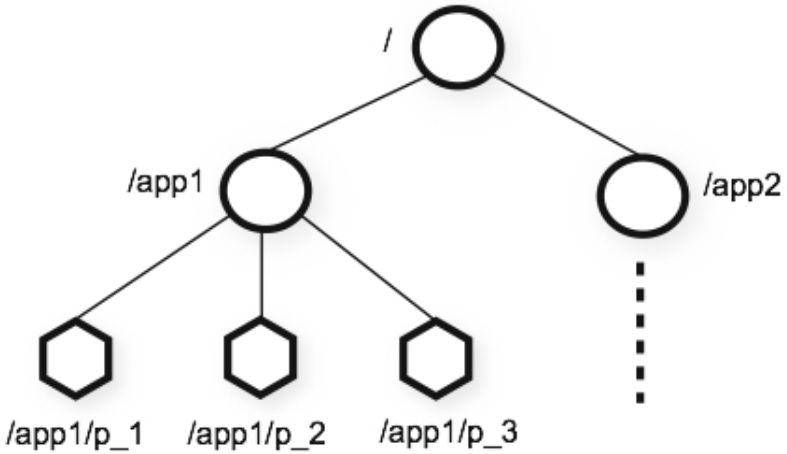


ZooKeeper: Hierarchischer Namesraum als Store



- ZooKeeper implementiert hierarchischen Namensraum mittels einer Baumdatenstruktur.
- Nodes des Baums sind können Informationen (bytes) und/oder weitere Kinder haben.
- Die Nodes und sind PERSISTENT oder EPHEMERAL:
  - PERSISTENT: Überdauern Verbindungsabbruch durch Client.
  - EPHEMERAL: Gelöscht nach Verbindungsabbruch durch Client.

ZooKeeper: Beispieloperationen auf Store



– Erstelle persistenten Node /app1/p_3 mit Inhalt myData (byte-Array) und erlaube Zugriff von allen Nodes (`Ids.`*`OPEN_ACL_UNSAFE`*`):`

```
zk.create("/app1/p_3", myData, Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT);
```

– Frage Inhalt von Node /app2 ab (nachträgliche Änderungen, lösen Callback `setChanged()` aus, in `stat` werden Statistiken geschrieben.

```
byte[] data = zk.getData("/app2", event -> setChanged(), stat);
```

Beispiel (Auszug): Leader-Election mittels ZooKeeper

```
public final static String PATH = "/myAppLeader";
public final String instance_name = ... ;  // unique instance name
public String primary_name = null; // primary name (not known at startup)

public void getOrBecomeLeader() {
   try {

      zk.create(PATH, instance_name.getBytes(), Ids.OPEN_ACL_UNSAFE,
                CreateMode.EPHEMERAL);
      primary_name = instance_name;

   } catch (KeeperException.NodeExistsException e) {
      try {
         byte[] data = zk.getData(PATH, event->getOrBecomeLeader(), stat);
         primary_name = new String(data);
      } catch (KeeperException.NoNodeException ex) {
         getOrBecomeLeader(); // leader has changed
      }

   } catch (KeeperException | InterruptedException ex) {
      throw new RuntimeException(ex); // must not occur
}}
```

Erstelle an Verbindung gebundenen Node. Falls erfolgreich: werde Primary.

Anderfalls: Node existiert: Frage Inhalt (PrimaryName) ab

Zusammenfassung

- CAP-Theorem zeigt Grenzen für verteilte Systeme auf bzgl. Konsistenz, Verfügbarkeit und Partitionstoleranz.
- Konsistenz: Alle beteiligen Systeme haben identische und aktuelle Daten.
- Konsistenzgarantien:
  - Sequentielle Konsistenz bedingt eine zeitliche Kopplung der Systeme.
  - Eventual Consistency erlaubt Systeme zu entkoppeln.
- Replikation:
  - Primary-Backup-Protokoll für on-the-fly Replikation.
  - Standardimplementationen mit verschiedenen Variationen.
- Ausfallsicherheit mittels Replikas:
  - Erkennung des Ausfalls des Primary mittels Heartbeat.
  - Wahl des neuen Primary: Entweder mittels Consensus oder zentralem Store.

Literatur

- Distributed Systems (3rd Edition), Maarten van Steen, Andrew S. Tanenbaum, Verleger: Maarten van Steen (ehemals Pearson Education Inc.), 2017.

Fragen?