

Verteilte Systeme und Komponenten

Fehlertoleranz und Resilienz

Martin Bättig

Letzte Aktualisierung: 4. November 2022

FH Zentralschweiz



Inhalt

- Definitionen und Begriffe
- Fehlertolerante Systeme
- Resilienz durch Wiederherstellbarkeit
- Verteilte Transaktionen

Lernziele

- Sie kennen allgemeine Begriffe und Definitionen im Zusammenhang mit Fehlertoleranz und Resilienz.
- Sie wissen, an welchen Stellen beim Senden und Empfangen einer Message Fehler auftreten können.
- Sie wissen, was Idempotenz ist und wie Sie diese bei der Auslieferung von Messages ausnutzen können.
- Sie kennen die verschiedenen Ausliefergarantien für Messages.
- Sie kennen die Möglichkeiten um eine exactly-once Auslieferung zu realisieren.
- Sie verstehen das Prinzip der verteilten Transaktionen und ihre Anwendung bei der messageorientierten Kommunikation.

Definitionen und Begriffe

Definitionen

- **Verfügbarkeit:** System für sofortigen Einsatz bereit. Beschreibt einen Zeitpunkt. Ausgefallene oder ausgeschaltete Systeme sind nicht verfügbar.
- **Hochverfügbarkeit:** System ist mit sehr hoher Wahrscheinlichkeit für Einsatz bereit.
- **Zuverlässigkeit:** Zeitintervall innerhalb dessen ein System verfügbar ist.
- **Betriebssicherheit:** Gibt an, ob ein Ausfall eines Systems zu katastrophalen Ereignissen führen kann.
- **Wartbarkeit:** Wie schnell kann ein ausgefallenes System wieder hochgefahren werden.

Definitionen

Fehler: Ursache einer Funktionsstörung.

Fehlertoleranz: System kann trotz Vorkommnissen von Fehlern seine Dienste anbieten.

Resilienz: Widerstandfähigkeit gegenüber vorhersehbaren bekannten Fehlern.

Resilienz durch Redundanz:

- ⇒ Schutz gegen Systemausfall: Systeme oder Daten mehrfach vorhanden. (wird im Teil «Konsistenz und Replikation» behandelt).
- ⇒ Schutz gegen byzantinische Fehler (korrupte Daten) -> erfordert Consensus-Protokolle (komplexes Thema, nur kurze Einführung).

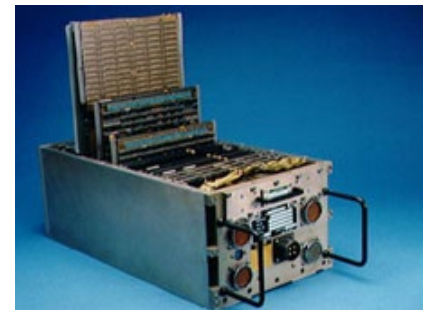
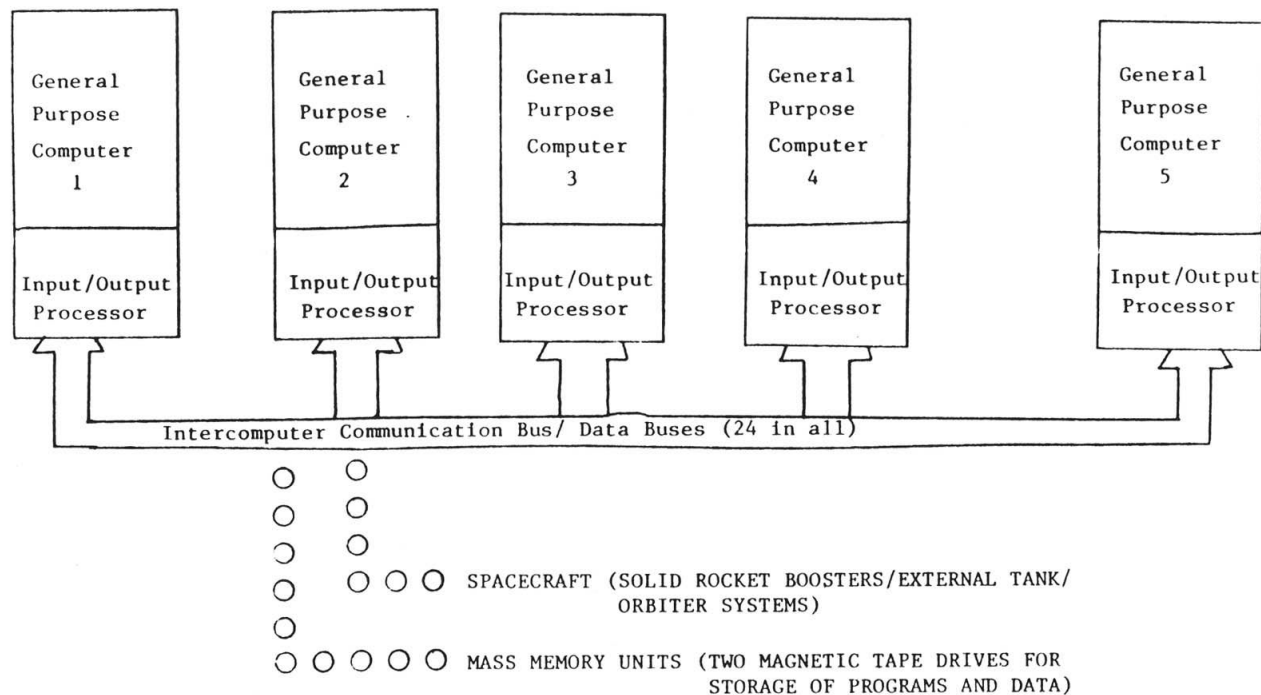
Resilienz durch Wiederherstellbarkeit:

- ⇒ System kann nach Ausfall in kurzer Zeit wiederhergestellt werden und operiert exakt wie vor dem Ausfall (-> Wartbarkeit).

Consensus-Protokolle

Consensus (dt. Konsens): Bei unterschiedlichen Ausgaben redundanter Systeme muss festgelegt werden, welche Ausgabe verwendet wird.

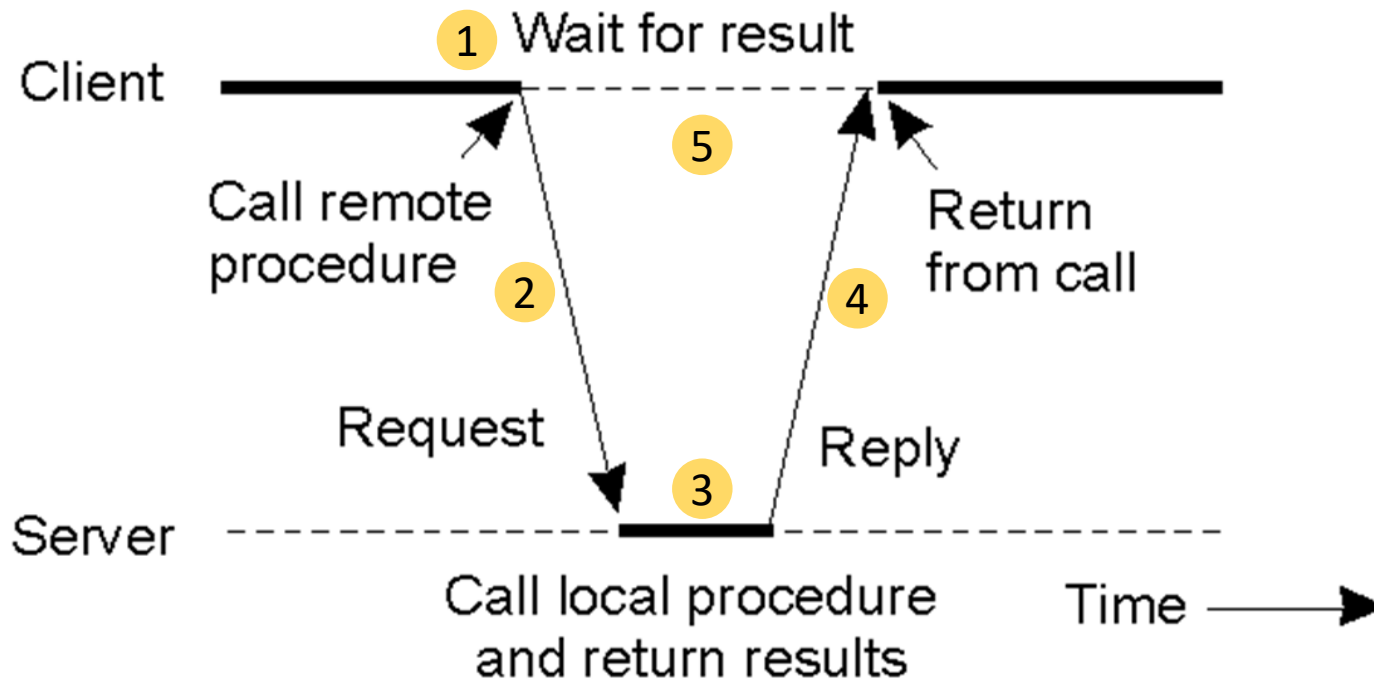
Beispiel: Vernetzte Steuerungscomputer des Space-Shuttles: Vier identische Computer und einer mit unterschiedlicher Programmierung.



Fehlerarten bei verteilten System

- **Server nicht erreichbar:**
 - Daten / Messages können nicht verschickt werden.
- **Auslassungsfehler: Falsche Reihenfolge der Daten / Messages:**
 - Von TCP zuverlässig verhindert (mit welchem Mechanismus?).
- **Abrupter Verbindungsabbruch durch Netzwerkausfall:**
 - Folge: Verlorene Daten (z.B. Messages).
- **Abrupter Verbindungsabbruch durch Systemausfall:**
 - Folge: Verlorene Daten (z.B. Messages) oder inkonsistente Daten.

Potentielle Fehler bei einem synchronen Aufruf (z.B. RPC)

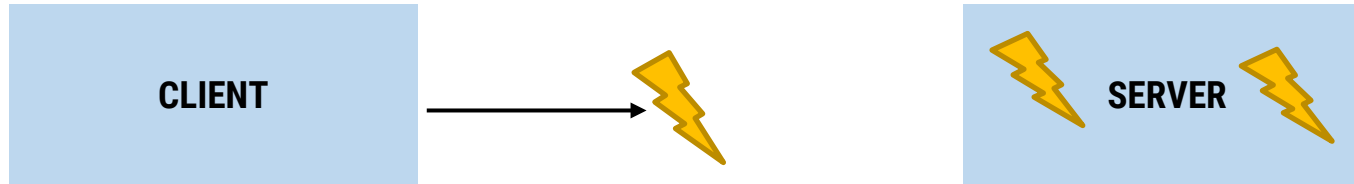


1. Keine Verbindung
2. Request geht verloren
3. Server crasht
4. Reply geht verloren
5. Client crasht

Fehlertolerante Systeme

Fehlersituation: Server nicht erreichbar (ohne Redundanz)

- Die Gegenstelle ist nicht erreichbar **VOR** Absenden einer Message:



Entweder Verbindung unterbrochen oder Server läuft nicht (identischer Effekt für Client).

Erkennung: Aufbau der Verbindung schlägt fehl.

Massnahmen:

- Kein Betrieb möglich (z.B. Terminal-Server) => User informieren.
- Reduzierter Betrieb, falls Server nur Zusatzfunktionalität anbietet.
- Verwendung eines lokalen Caches (Zwischenspeicher).

Verwendung eines lokalen Caches

- **Beim Lesen:** Beziehe Information aus vorangehender Kommunikation.
 - **Beim Schreiben:** Führe Schreib-Operationen lokal durch, sende an Server sobald verfügbar.
- ⇒ Achtung «Merge-Konflikte» analog z.B. VCS, falls mehrere Teilnehmer oder Geräte Änderungen vornehmen können.

EchoClient mit lokalem Cache: CachingEchoHandler als Thread

```
public static class CachingEchoHandler implements Runnable {  
    Queue<String> queue = new ConcurrentLinkedQueue<>();  
    ZContext context;  
    ZMQ.Socket socket;
```

Erstelle lokale Queue




```
    public CachingEchoHandler() {  
        context = new ZContext();  
        socket = context.createSocket(SocketType.REQ);  
        socket.connect(ADDRESS);  
    }
```

ZeroMQ-Socket erstellen



```
    public void run() {  
        try {  
            while (true) {  
                sendAndReceiveMessage(waitForNextMessage());  
            }  
        } catch (InterruptedException e) {  
            throw new RuntimeException(e);  
        }  
    }
```

Warte auf Message in lokaler Queue.



EchoClient mit lokalem Cache: Queue and Sending

```
private void sendAndReceiveMessage(String message) {  
    socket.send(message.getBytes(ZMQ.CHARSET));  
    byte[] bytes = socket.recv();  
    System.out.println(new String(bytes, ZMQ.CHARSET));  
}
```

Message senden
und empfangen
(blockierend)

```
public void addMessageToQueue(String s) {  
    queue.offer(s);  
    synchronized (this) { this.notify(); }  
}
```

Message zu lokaler Queue
hinzufügen

```
private String waitForNextMessage() throws InterruptedException {  
    String message;  
    while ((message = queue.poll()) == null) {  
        synchronized (this) { this.wait(); }  
    }  
    return message;  
}
```

Warte bis neue
Message in lokaler
Queue eintrifft.

CachingEchoClient - Main

```
public static void main(String[] args) throws IOException {  
    CachingClientHandler cachingEchoHandler = new CachingClientHandler();
```

Starte ClientEchoHandler als Thread, da Empfang von Messages blockieren wird, falls EchoServer nicht verfügbar.

```
new Thread(cachingEchoHandler).start();
```



```
LOG.info("CachingEchoClient running on " + ADDRESS);
```

```
BufferedReader userIn =
```

```
    new BufferedReader(new InputStreamReader(System.in));
```

```
while(true) {
```

```
    String input = userIn.readLine();
```

```
    cachingClientHandler.addToQueue(input);
```

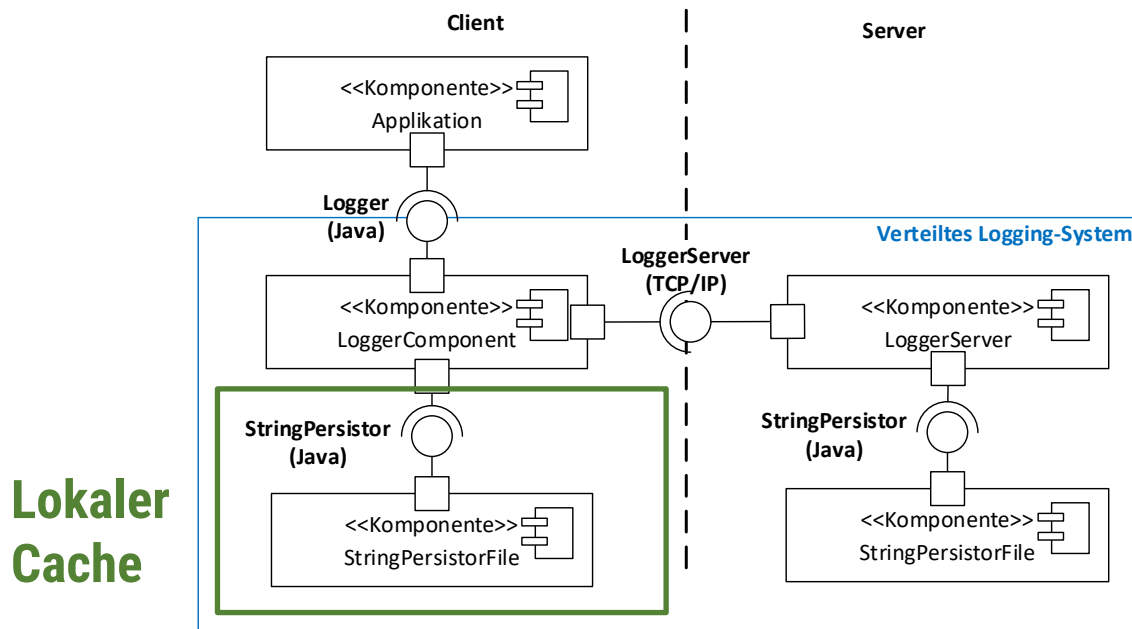
```
}
```

```
}
```

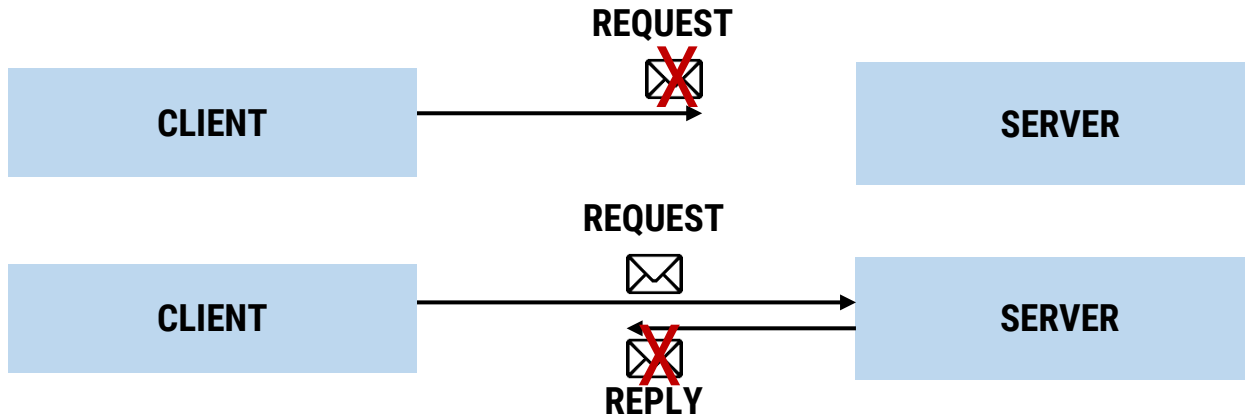
Übung: Lokaler Cache im verteilten Logger-System

Diskutieren Sie in Ihrem Team, wie Sie mittels einem lokalen Cache mit folgenden Situationen möglichst transparent umgehen können:

- 1) Fehlgeschlagener initialer Verbindungsaufbau, nach ca. 10 Sekunden steht die Verbindung.
- 2) Fehlgeschlagener initialer Verbindungsaufbau, Server kommt während des ganzen Logging-Vorgangs nicht mehr online.
- 3) Abrupter Verbindungsabbruch, mit Wiederherstellung der Verbindung nach ca. 10 Sekunden. Messages können verloren gehen.



Fehlersituation: Verlorene Message (Request oder Reply)



Erkennung:

- Client startet Timer bei Absenden eines Requests.
- Ist nach Ablauf des Timers noch keine Antwort eingetroffen, gilt die Message als verloren.
- Unterscheidung eines verlorenen Requests von einem verlorenen Reply?

Massnahmen:

- Benutzer Informieren / Fragen z.B. bei interaktiven Verbindungen.
- Falls sinnvoll, Request nochmals senden => ggf. Duplikatscheck

Auslieferungsgarantien

- **At-least-once:** Message wird sooft gesendet bis eine Antwort eintrifft.
 - Problem: Aktion wird möglicherweise doppelt ausgeführt.
- **At-most-once:** Message wird höchstens einmal gesendet.
 - Problem: Aktion wird möglicherweise nicht ausgeführt.
- **Exactly-once:** Message wird exakt einmal gesendet.
 - Gewünscht, aber möglicherweise zu teuer oder unnötig.

Transparenz mittels Idempotenz

Idempotente Funktion (math.):

Funktion, welche auf sich selbst angewandt das identische Resultat ergibt:

$$f(x) == f(f(x))$$

Beispiel für eine idempotente Funktion:

- Rückgabe des absoluten Werts: $f(x) = |x|$

Beispiel für eine nicht-idempotente Funktion:

- Rückgabe der Negation: $f(x) = -x$

Idempotente Anfragen

- Anfragen sind idempotent, wenn die Funktion auf der Gegenseite idempotent ist.
- Wichtig: eine idempotente Anfrage darf keine Nebeneffekte wirken.
- Idempotente Anfragen können i.d.R. ohne Probleme wiederholt werden!

Beispiele für idempotente Anfragen

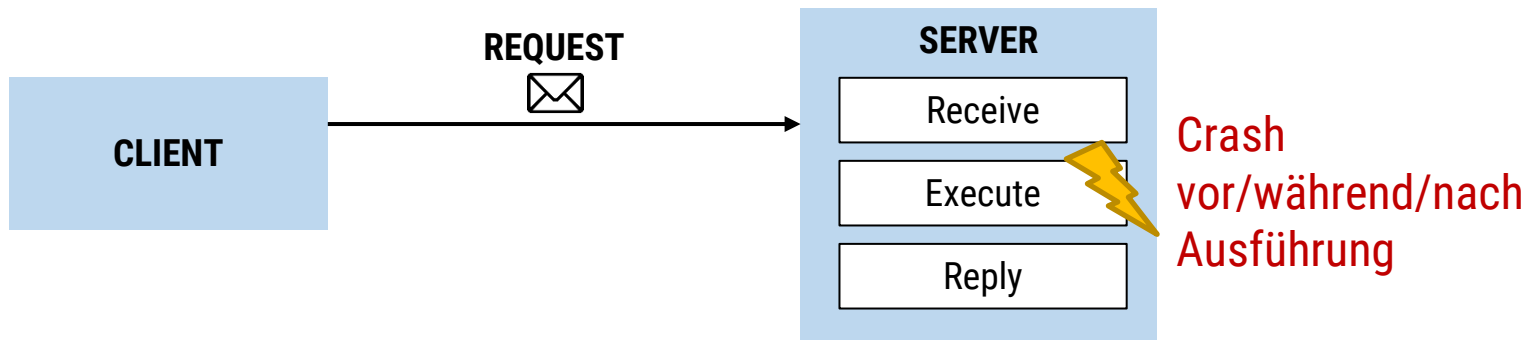
- Read-Requests: z.B. Abfrage nach einem Fahrplan.
 - Achtung: Falls Lese-Zugriffe protokolliert werden (z.B. bei Banken).
- Ändere Adresse von Ort A zu Ort B (ohne weitere Nebeneffekte).
 - Achtung bei parallelen Anpassungen, ggf. Zeit mitsenden.

Nicht idempotent:

- Bestellung / Reservation / etc. (typisch: "erstelle neues Element").
- Lösche File F, wenn F in Zwischenzeit wieder erstellt werden kann.

Resilienz durch Wiederherstellbarkeit

Fehlersituation: Server-Crash während Requestverarbeitung



Erkennung:

- Client: Keine Unterscheidung gegenüber verlorener Message möglich.
- Server kann über Aktionen ein Log führen und dieses beim Restart abarbeiten (zusätzliche Kosten durch Diskzugriff).

Massnahme:

- Falls vor Ausführung (== verlorener Request): Client kann Message erneut senden.
- Falls während Ausführung: Konsistenz wieder herstellen.
- Falls nach Ausführung: Reply-Senden.
 - **Achtung:** Client könnte erneute Anfrage gesendet haben => **Duplikat**.

Fehlersituation: Client-Crash während Warten auf Antwort



Entweder Verbindung unterbrochen oder Client läuft nicht (identischer Effekt).

Erkennung:

- Server: Keine Verbindung zu Client möglich.

Massnahme:

- Antwort speichern, falls Client identifizierbar (=> z.B. mittels Token).
 - **Achtung:** Client könnte erneuten Request stellen => Duplikatserkennung.
- Antwort verwerfen, falls Client nicht identifizierbar.
- Problematisch, z.B. falls ein Client Ressourcen blockieren kann.
Beispiel: File-Locking im NFS (Network File System).

Herausforderung: Erkennung von Duplikaten

Entweder durch:

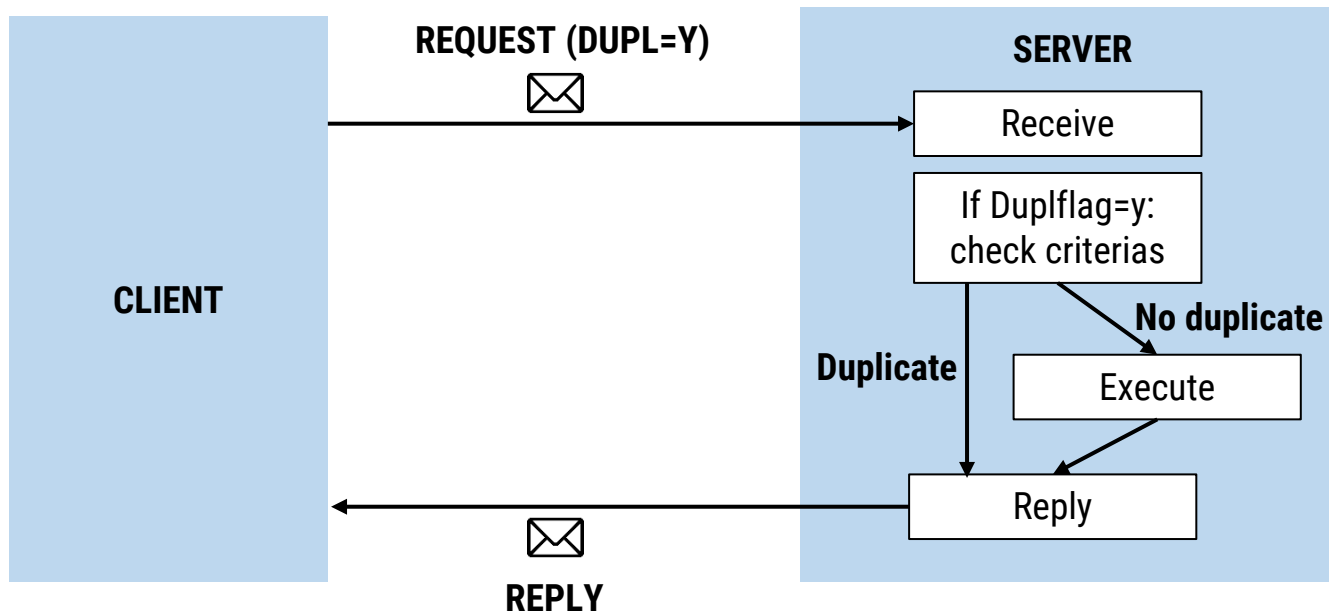
- **Heuristik (*)**: falls eine gewisse Fehlerwahrscheinlichkeit tolerierbar ist.
- **Sequenznummer**: falls alle Duplikate erkannt werden sollen.

(*) Heuristik: Methode um mittels **unvollständigem Wissen** trotzdem zu praktikablen Ergebnissen zu kommen.

Erkennung von Duplikaten mittels Heuristik

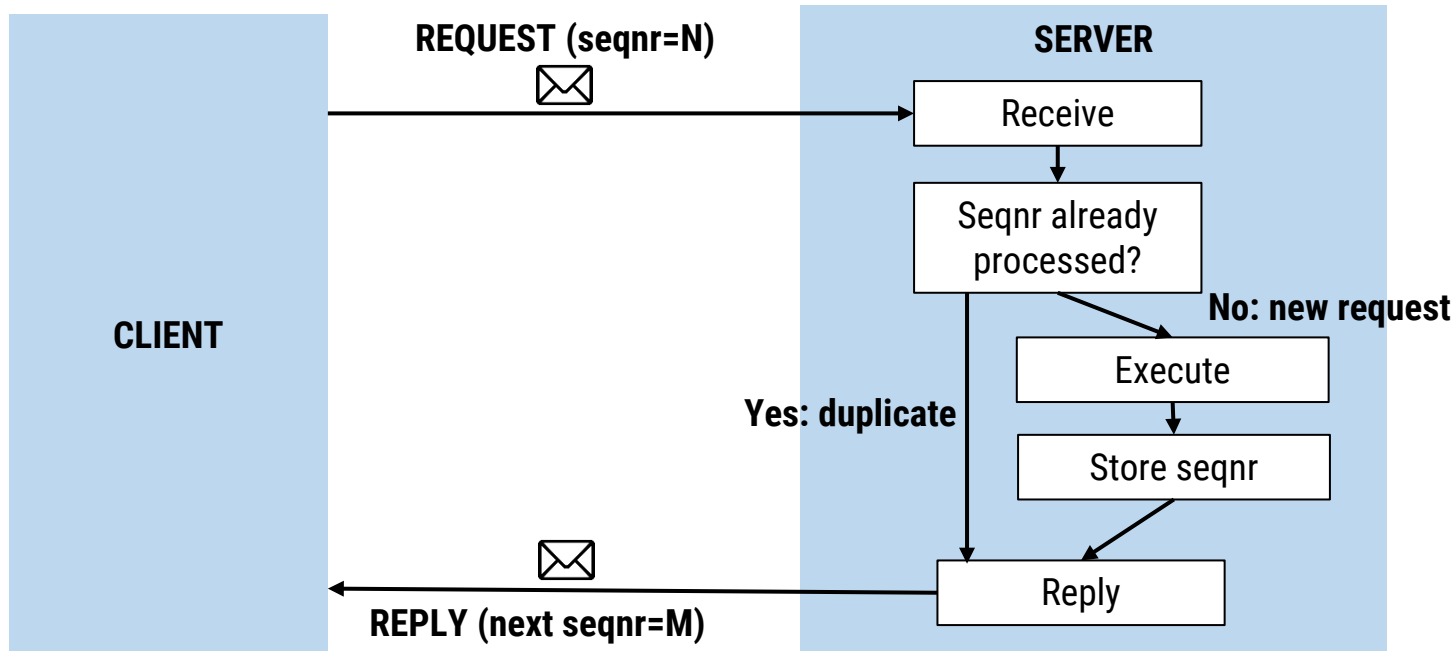
Mit gewisser Wahrscheinlichkeit durch Heuristiken:

- Duplikatsflag: Client gibt an, dass ein Request wiederholt wird.
- Verwendung diverser Kriterien (Kundennummer, Betrag, Ort, usw.).



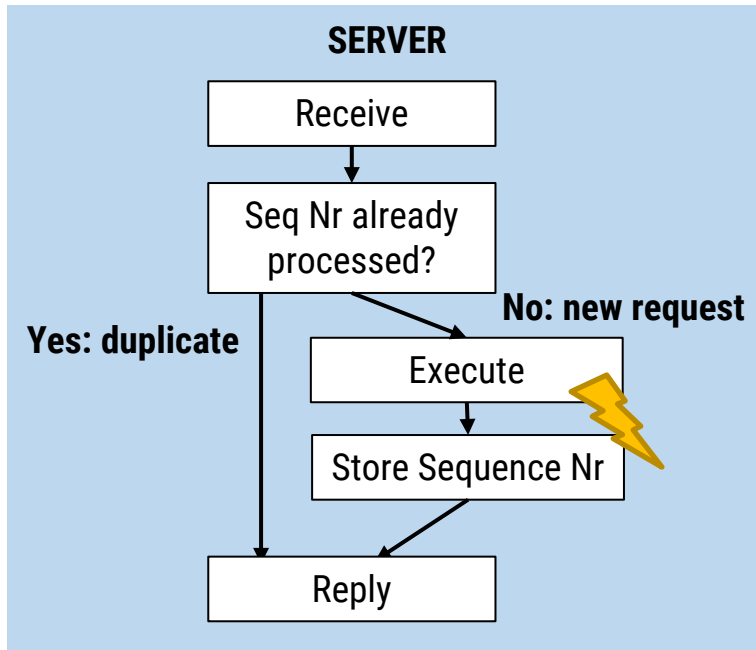
Exakte Erkennung von Duplikaten mittels Sequenznummern

- Jeder Request erhält eine Sequenznummer.
- Server muss sich nach oder vor Ausführung die Sequenznummer (seqnr) merken (Kosten: Zugriff auf persistenten Speicher).
- I.d.R. grosse Nummer -> erhöht Grösse der Messages.
- **Mögliches Vorgehen:** Server wird bei jeder Antwort die nächste Sequenznummer vorgeben (Request-Response).

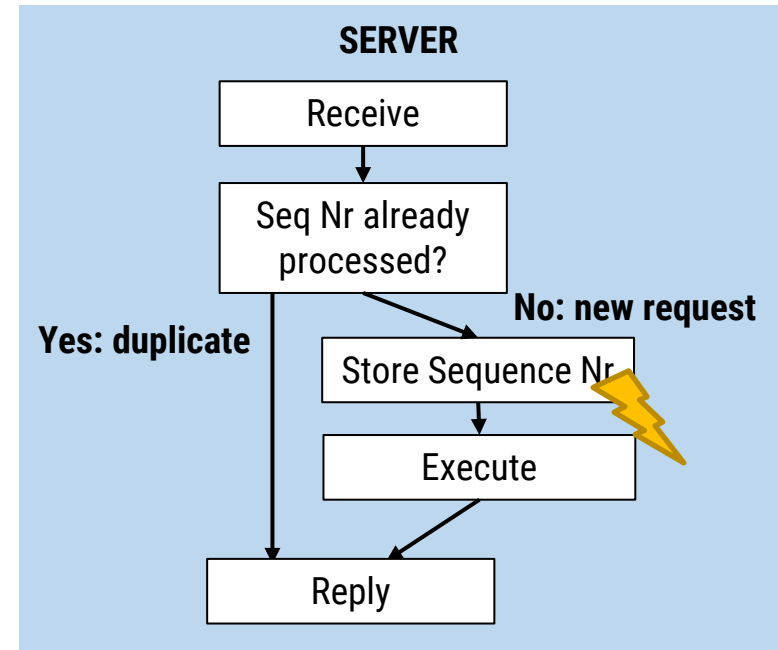


Server-Crash: Fehlende Konsistenz

- Wie sicherstellen, dass Sequenznummer N nur als verarbeitet markiert wird, wenn der Request ausgeführt wurde?
- Andere Reihenfolge hilft nicht:



Crash vor "Store Sequence Nr":
Aktion ausgeführt, aber nicht als ausgeführt markiert.
=> Doppelte Ausführung



Crash vor Execute:
Nicht ausgeführt, aber als ausgeführt markiert.
=> Keine Ausführung

Transaktionen zur Lösung des Konsistenzproblems

Transaktion:

- Atomare Einheit der Ausführung: Entweder alles ausgeführt oder nichts.
- Typischerweise innerhalb von Datenbanken angewendet und unterstützt (Oracle / Postgres / MariaDB / H2 / DB2 / MongoDB / etc.).

Ablauf:

- Transaktion starten.
 - Mehrere zusammengehörige Operationen innerhalb der Transaktion durchführen (Daten abfragen, einfügen, modifizieren, löschen).
 - Transaktion entweder:
 - **erfolgreich abschliessen (commit)**: Alle Operationen werden ausgeführt
- ODER**
- **abbrechen (rollback)**: Keine Operation wird ausgeführt.

Transaktionen zur Lösung des Konsistenzproblems (forts.)

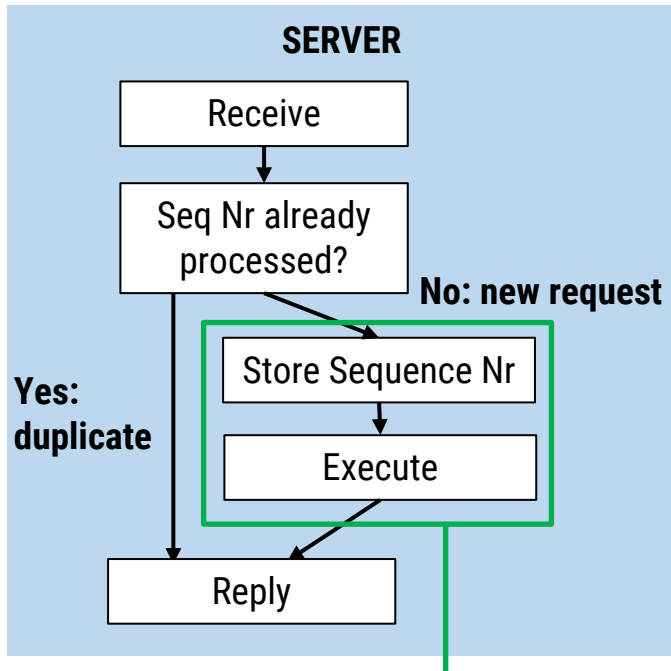
Beispiel: Geldtransfer von 100 CHF von Konto A zu Konto B.

1. Transaktion T starten
2. Account A um 100 CHF reduzieren
3. Account B um 100 CHF erhöhen
4. commit von T

Falls Crash vor Schritt 4 => Keine Aktion ausgeführt.

Lösung des Konsistenzproblems mit Transaktionen

Vorgehen: Verwendung einer Datenbank D, welche Transaktionen unterstützt.



Ausführung als Transaktion:
Entweder beide Aktionen
ausgeführt oder keine.

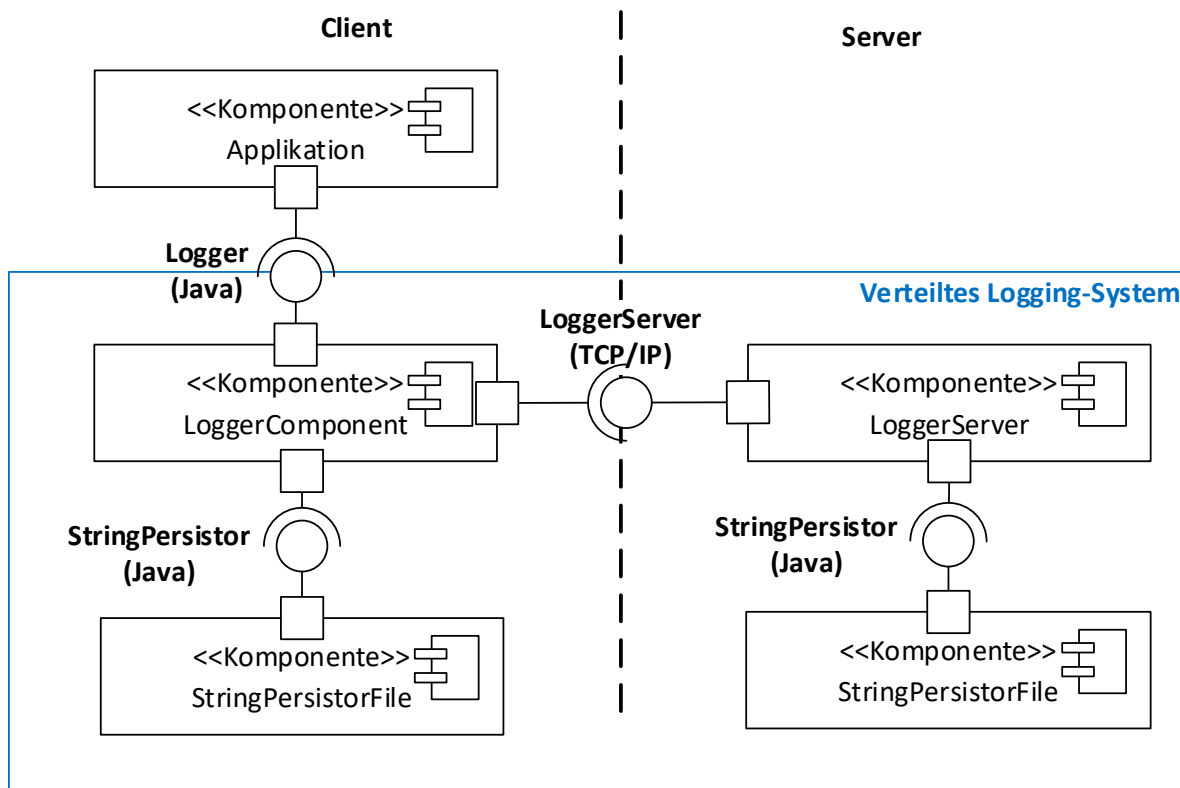
Vorgehen (Skizze):

1. Starte Transaktion T.
2. Verarbeitung speichert alle Änderungen in Datenbank D.
3. Speichere Sequenznummer in Datenbank D.
4. Schliesse Transaktion T ab.

Übung: Exactly-once Auslieferungsgarantie im verteilten Logger

Diskutieren Sie in Ihrem Team, wie Sie eine Exactly-once Auslieferungsgarantie im verteilten Logger-System realisieren könnten (Hypothetisch, keine Anforderung):

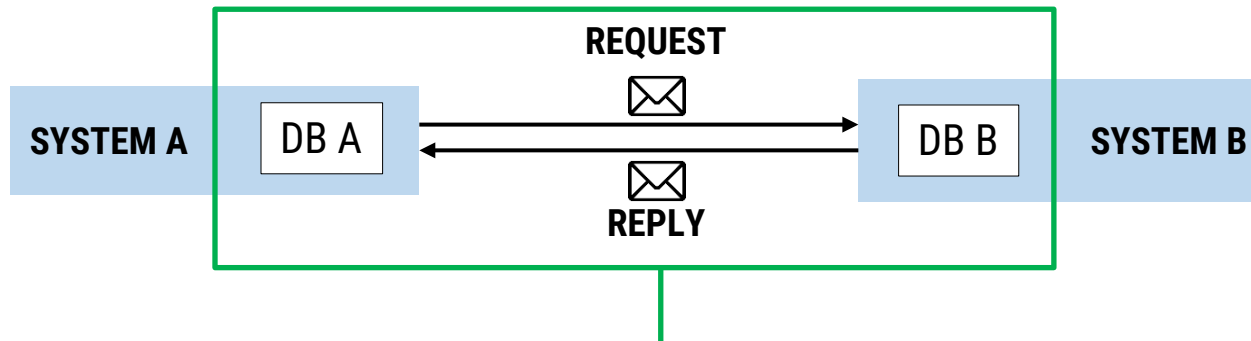
- 1) Wie müssten Sie das Protokoll ändern?
- 2) Hätte dies Einfluss auf die Komponentenaufteilung oder Schnittstellen?



Verteilte Transaktionen

Verteilte Transaktionen

- **Ziel:** Kombination von Transaktionen über zwei oder mehr Systeme hinweg. Alle Transaktionen werden entweder ausgeführt oder nicht.
- **Voraussetzung:** Jedes an der verteilten Transaktion teilnehmende System verfügt über einen Transaktions-Mechanismus, z.B. eine Datenbank (DB).



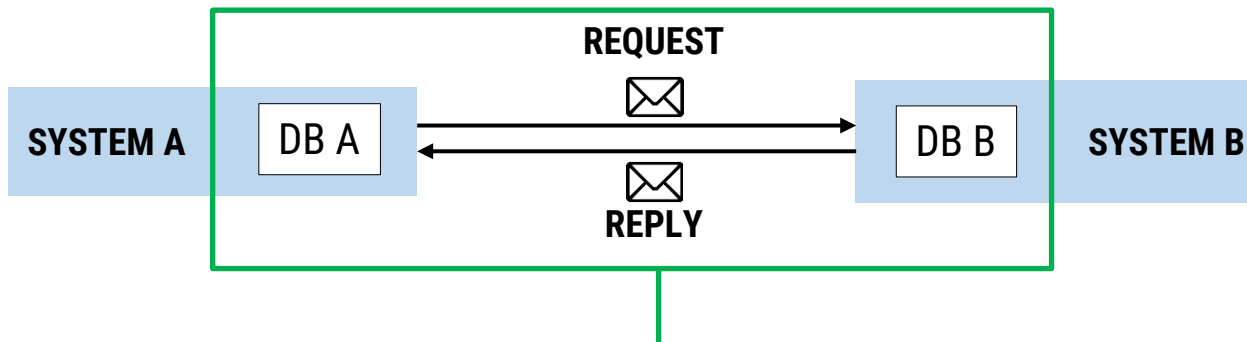
Verteilte Transaktion: Entweder beides ausgeführt oder nichts.

Weiterführende Literatur:

Van Steen/Tannenbaum, Distributed Systems, Kapitel 8.5

Kommunikation mit verteilten Transaktionen

- **Vorteil:** Einsatz von Sequenznummern usw. ist unnötig.
- ⇒ **Nachteil:** Höherer Ressourcenbedarf und Administrationsaufwand.
- ⇒ **Achtung:** Man muss transaktional programmieren (kein autocommit!).
- ⇒ **Achtung:** Bei System-Crashes können beide Systeme blockiert werden.



Verteilte Transaktion: Entweder beides ausgeführt oder nichts.

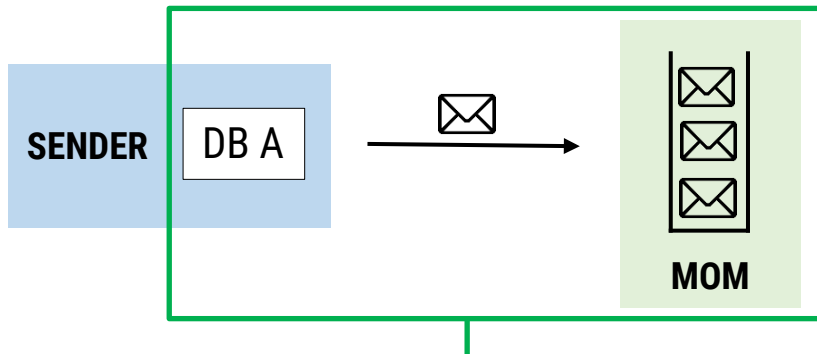
Skizze:

System A: delete msg from outgoing where id = X

System B: insert (msg) into incoming

Verteilte Transaktionen mit persistenter Kommunikation

- Message-orientierte Middleware (MOM) unterstützt oft verteilte Transaktionen.
- **Ziel:** Zeitliche Entkopplung mit *exactly-once* Auslieferungsgarantie.



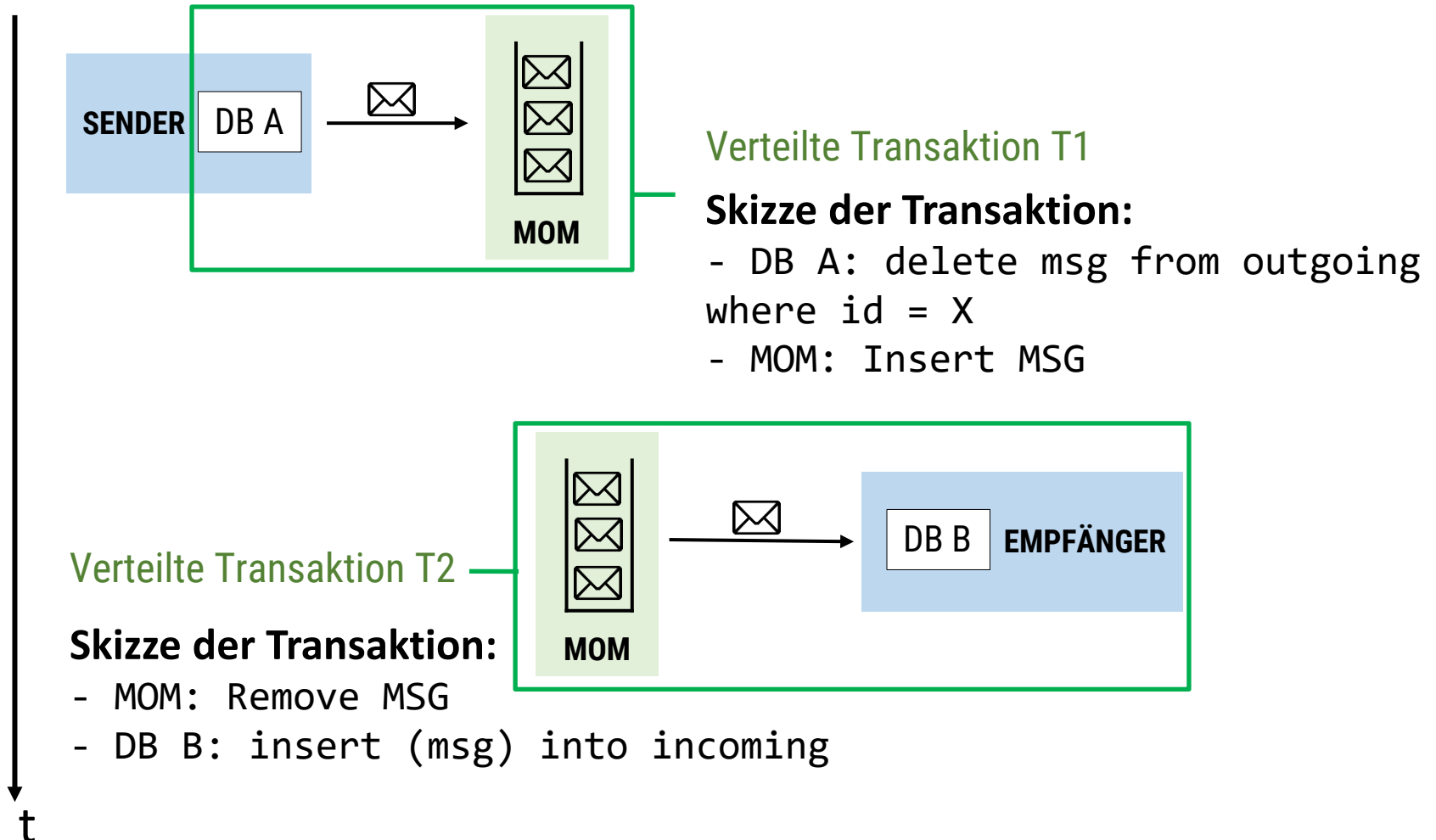
Verteilte Transaktion T1

Skizze der Transaktion:

- DB A: delete msg from outgoing where id = X
- MOM: Insert MSG

Beispiel (Skizze): Exactly-once Auslieferung mittels MOM

- Zuerst wird T1 ausgeführt, dann (ggf. Minuten später) T2:

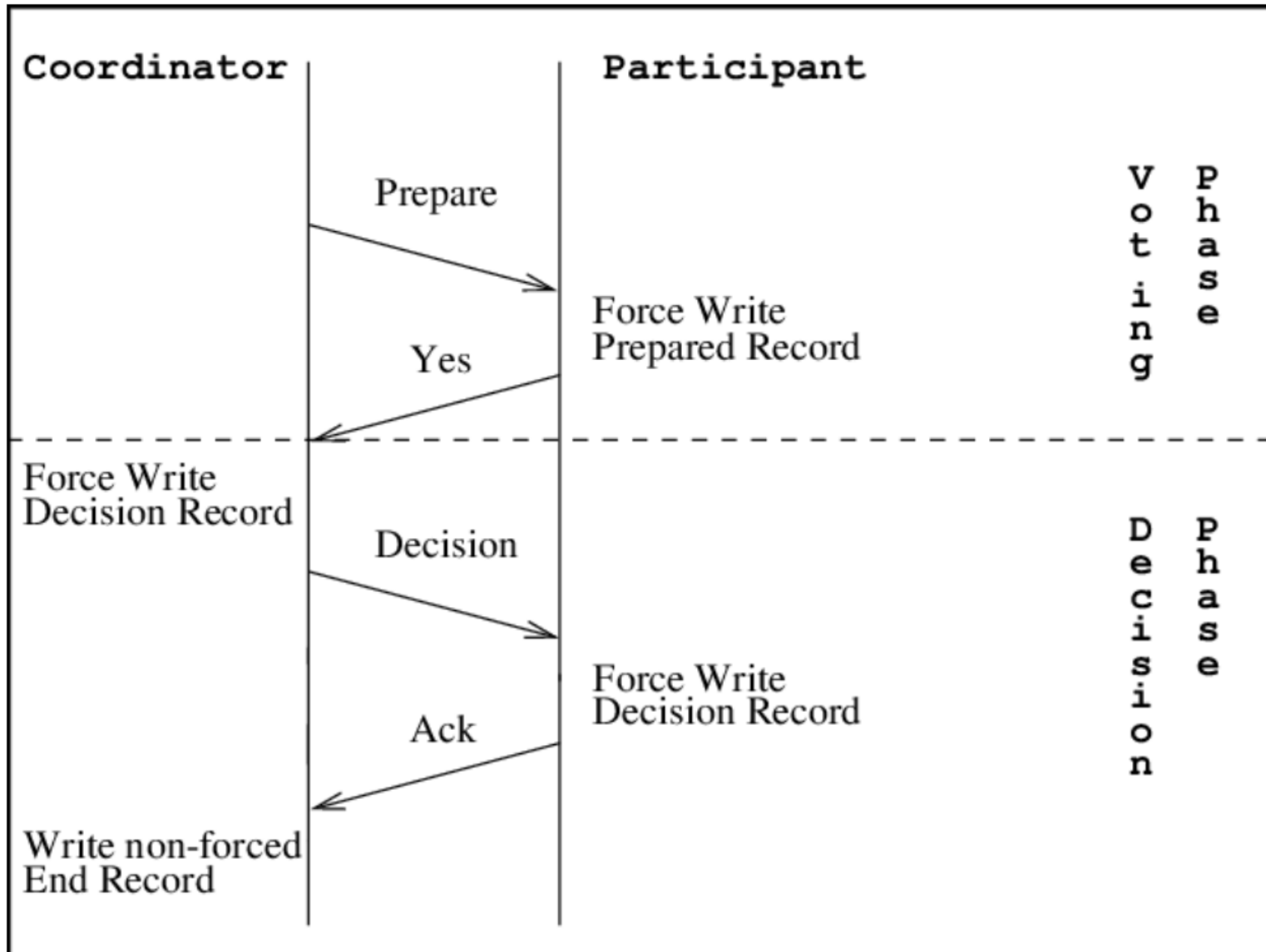


Funktionsweise verteilter Transaktionen

Two-Phase-Commit (2PC): Typischer Algorithmus für verteilte Transaktionen.

- Koordiniert wird 2PC vom einem Teilnehmer der verteilten Transaktion, z.B. der ersten Transaktion, welche ein Commit ausführt.
- **Erste Phase:** Koordinator fragt alle beteiligten Systeme, ob Sie die Transaktion erfolgreich abschliessen können (YES) oder nicht (NO).
- **Zweite Phase:** Koordinator trifft Entscheidung: Entweder alle commiten (nur YES erhalten), oder alle brechen die Transaktion ab (mindestens ein NO erhalten).

Two-Phase-Commit: Ablauf



Two-Phase Commit im Detail

Voting Phase:

- Koordinator sendet eine PREPARE-Anfrage an alle Teilnehmer.
- Falls ein Teilnehmer eine PREPARE-Anfrage erhält, antwortet er entweder mit YES, falls er seine Transaktion abschliessen kann und will, ansonsten antwortet er mit NO.

Decision Phase:

- Koordinator sammelt alle Antworten der Teilnehmer.
 - Falls alle Teilnehmer mit YES geantwortet haben, sendet er eine COMMIT-Anfrage an alle Teilnehmer.
 - Falls mindestens ein Teilnehmer mit NO geantwortet hat, sendet er eine ABORT-Anfrage an alle Teilnehmer.
- Jeder Teilnehmer, welcher mit YES geantwortet hat, wartet für auf die Anweisung des Koordinators.
 - Falls er eine COMMIT-Anfrage erhält, schliesst er die Transaktion erfolgreich ab.
 - Falls er eine ABORT-Anfrage erhält, mache Transaktion rückgängig.
- Falls Koordinator crasht, können andere Teilnehmer angefragt werden.

Zusammenfassung

- Fehlertoleranz: System kann bestimmte Fehler tolerieren.
- Beim Senden einer Message können eine Vielzahl an Fehlern auftreten.
- Idempotente Messages können erneut gesendet werden.
- Auslieferungsgarantien je nach Anwendungsfall wählen: Exactly-once ist schwierig.
- Erkennung von Duplikaten für exactly-once Auslieferungsgarantie mittels Sequenznummer.
- Verteilte Transaktionen ermöglichen eine exactly-once Auslieferungsgarantie der Messagekommunikation und können mit Message-Oriented-Middleware kombiniert werden.
- Basis der verteilten Transaktionen ist der Two-Phase-Commit.

Literatur

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

Fragen?