Hochschule RheinMain

FEHLERTOLERANTE SYSTEME

Verteilte Systeme

Prof. Dr. Georg Hinkel
24.05.2024

GLIEDERUNG

Datum	Vorlesung	Übungsblatt	Abgabe
19.04.2024	Einführung	HamsterLib	06.05.2024
26.04.2024	Netzwerkprogrammierung	Theorie	
03.05.2024	World Wide Web	HamsterRPC 1	20.05.2024
10.05.2024	Remote Procedure Calls	Theorie	
17.05.2024	Webservices	HamsterRPC 2	03.06.2024
24.05.2024	Fehlertolerante Systeme	Theorie	
31.05.2024	Transportsicherheit	HamsterREST	17.06.2024
07.06.2024	Architekturen für Verteilte Systeme	Theorie	
14.06.2024	Internet der Dinge	HamsterIoT	01.07.2024
21.06.2024	Namen- und Verzeichnisdienste	Theorie	
28.06.2024	Authentifikation im Web	HamsterAuth	15.07.2024
05.07.2024	Infrastruktur für Verteilte Systeme	Theorie	
12.07.2024	Wrap-Up	HamsterCluster (Bonus)	16.08.2024

AGENDA UND LERNZIELE

Hochschule RheinMain

Agenda

- Muster für fehlertolerante Systeme
  - Retry, Retry and Wait
  - Circuit Breaker
  - Bulkhead
- Datenversionierung
- Verteilte Transaktionen

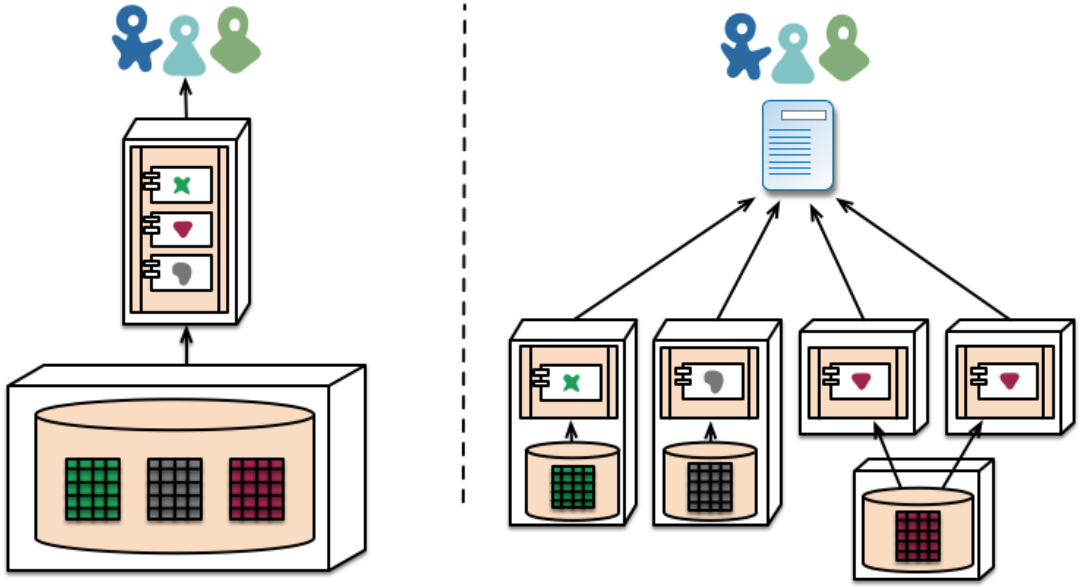
Lernziele

- Muster für verteilte Systeme anwenden können
- Datenversionierung anwenden können
- 2-Phasen-Commit-Protokoll anwenden können

MICROSERVICE-ARCHITEKTUREN

Hochschule RheinMain

- Anwendung wird aufgeteilt in mehrere Services, die unabhängig Teile der Funktionalität erbringen

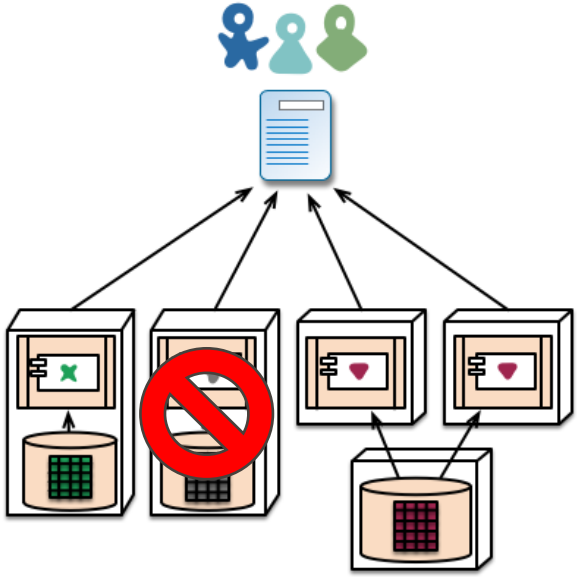


[https://martinfowler.com/articles/microservices.html]

RESILIENZ

Hochschule RheinMain

- ~ Widerstandsfähigkeit, Fehlertoleranz
- Wie reagieren, wenn ein (Micro-)Service ausfällt?



[https://martinfowler.com/articles/microservices.html]

Warum kann ein Dienst ausfallen?

Hochschule RheinMain

- 400 – Bad Request
- 401 – Unauthorized
- 403 – Forbidden
- 404 – Not Found
- 408 – Timeout
- …

Fehler eher im Verantwortungsbereich des Clients

- 500 – Internal Server Error
- 502 – Bad Gateway
- 503 – Service Unavailable
- 504 – Gateway Timeout

Fehler eher im Verantwortungsbereich des Servers

WAS KANN SCHON SCHIEF GEHEN?

Hochschule RheinMain

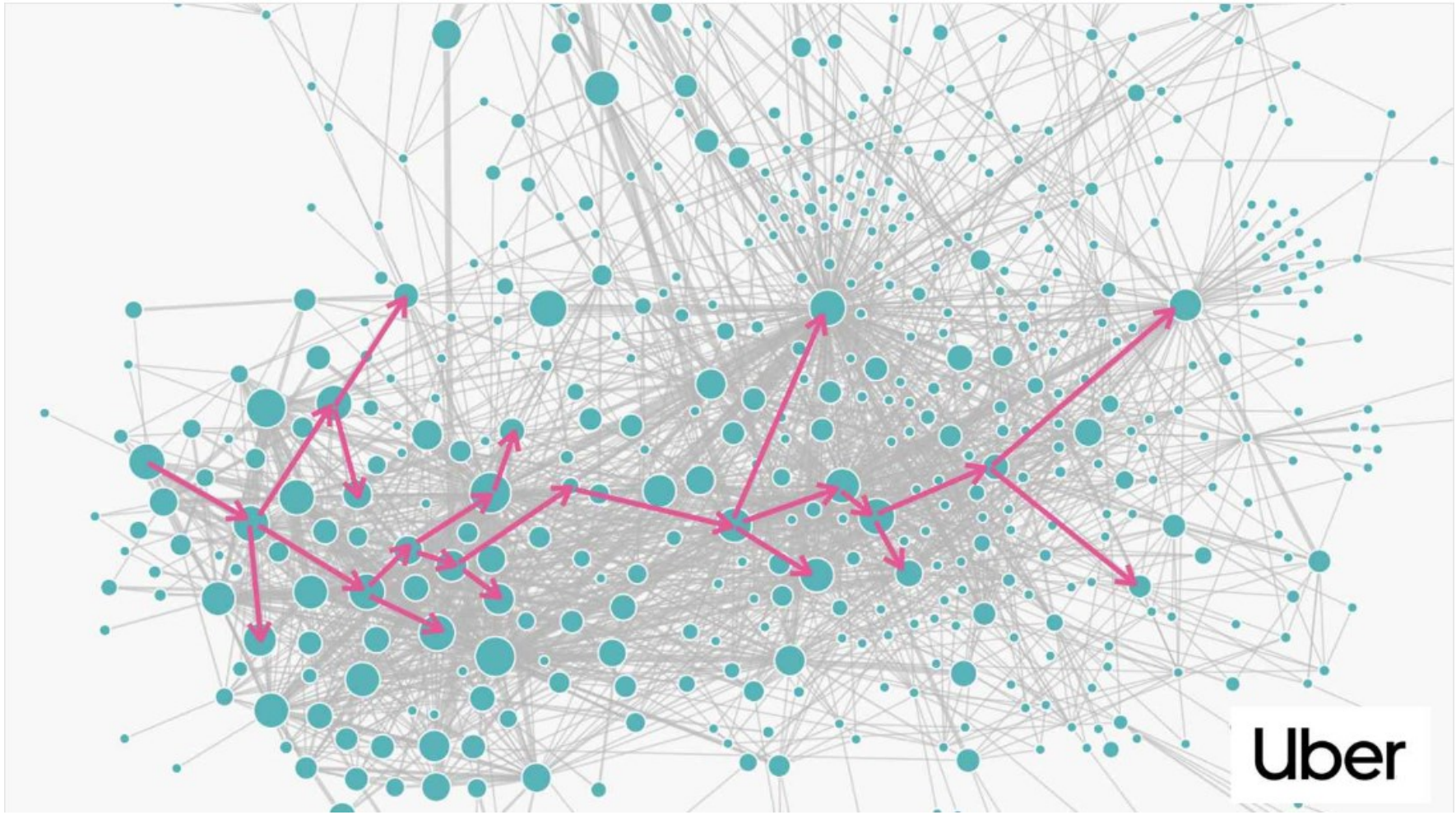
```
public class FlightSearch implements SessionBean {

  private MonitoredDataSource connectionPool;

  public List lookupByCity(. . .) throws SQLException, RemoteException {
    Connection conn = null;
    Statement stmt = null;

    try {
      conn = connectionPool.getConnection();
      stmt = conn.createStatement();

      // Do the lookup logic
      // return a list of results
    } finally {
      if (stmt != null) {
        stmt.close();
      }

      if (conn != null) {
        conn.close();
      }
    }
  }
}
```

- Katastrophaler Schaden
  - Kompletter Ausfall eines zentralen Dienstes einer großen Luftfahrtgesellschaft in den USA
  - Ausfall dauerte über 3h
  - Verlust von Reputation, Umsatz, …
- Code hatte alle qualitätssichernden Maßnahmen überstanden
  - Code Reviews
  - Unit Tests
  - Integration Tests
  - …

Fehler passieren ➔ Fokus auf Fehlertoleranz

MICROSERVICES BEI UBER

Hochschule RheinMain



[https://www.infoq.com/presentations/uber-microservices-distributed-tracing/]

RESILIENZMUSTER

Hochschule RheinMain

- Typische Lösungen für Fehlertoleranz in Mustern gekapselt
  - Ähnlich zu Entwurfsmustern aus der Softwaretechnik-Vorlesung
- Kapseln Lösungen für Probleme von fehlertoleranten Systemen

IMPLEMENTIERUNG VON PATTERNS

Hochschule RheinMain

- Patterns sind oft schon in Bibliotheken implementiert

- Beispiel für .NET: Polly
  - Open-source
  - Integration in ASP.NET Core
  - Fluent API

- Beispiel für Java: Resilience4j
  - Open-source
  - Integration in Spring
  - Steuerung über Annotationen

RETRY
Überblick

Hochschule RheinMain

- Nochmal probieren
- Begrenzung der Wiederholungsversuche

```
var policy = Policy.Handle<HttpRequestException>(httpExc => httpExc.StatusCode == HttpStatusCode.InternalServerError)
    .RetryAsync(3, onRetry: (exc, i) => _logger.LogWarning($"Attempt {i} failed with error {exc.Message}"));

var response = await policy.ExecuteAsync(() => _client.SendAsync(request));
```

RETRY
Einsatzzwecke

Hochschule RheinMain

- Zuverlässigkeit des Transportprotokolls
  - Paket erneut senden, wenn Empfang nach Ablauf eines Timeouts nicht bestätigt
  - Im Normalfall durch Protokolle abstrahiert

- Spontanes Versagen, Einzelfallprobleme
  - Internal Server Error
  - Timeouts
  - Deadlock Victim
  - …

RETRY AND WAIT
Überblick

Hochschule **RheinMain**

- Warten und nochmal nochmal probieren
- Begrenzung der Wiederholungsversuche
- Üblicherweise mit exponentiell ansteigender Wartezeit

```
var policy = Policy.Handle<HttpRequestException>(httpExc => httpExc.StatusCode == HttpStatusCode.InternalServerError)
    .WaitAndRetryAsync(3, i => TimeSpan.FromSeconds(Math.Pow(2, i)),
        onRetry: (exc, sleep) => _logger.LogWarning($"Request failed with error {exc.Message}, waiting {sleep}"));

var response = await policy.ExecuteAsync(() => _client.SendAsync(request));
```

```
RetryConfig config = RetryConfig.custom()
    .maxAttempts(2)
    .waitDuration(Duration.ofMillis(100))
    .retryOnResult(response -> response.getStatus() == 500)
    .retryOnException(e -> e instanceof WebServiceException)
    .retryExceptions(IOException.class, TimeoutException.class)
    .ignoreExceptions(BusinessException.class, OtherBusinessException.class)
    .build();
```

RETRY AND WAIT
Einsatzzwecke

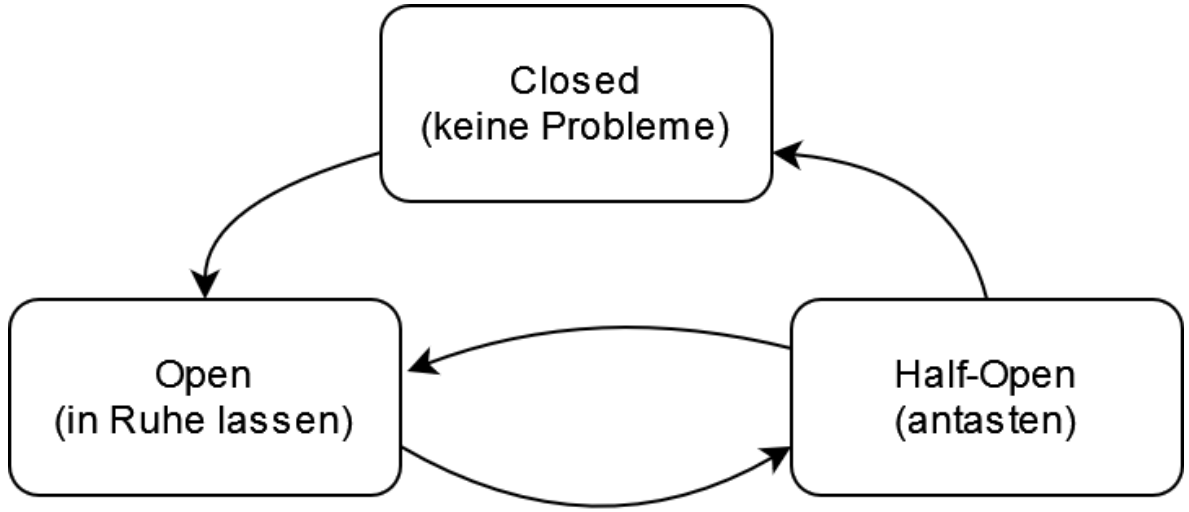
Hochschule RheinMain

- Temporale Probleme:
  - Gateway Timeout
  - Bad Gateway
  - Service Unavailable
  - Ggf. Unauthorized, not found
    - Ressource wurde evtl. gerade erst erstellt, Systemzustand noch nicht konsistent

CIRCUIT BREAKER
Überblick

Hochschule RheinMain

- “Die letzten Requests schlugen alle fehl, lassen wir den Service erstmal in Ruhe“
- Terminologie inspiriert von Schaltkreisen
  - Schaltkreis geschlossen: Requests werden ausgeführt
  - Schaltkreis offen: Requests werden gestoppt



CIRCUIT BREAKER
Implementierung in Polly / Resilience4J

Hochschule **RheinMain**

- Achtung: Instanzen müssen hier unbedingt erhalten werden

```
var policy = Policy.Handle<HttpRequestException>(httpExc => httpExc.StatusCode == HttpStatusCode.InternalServerError)
    .CircuitBreakerAsync(5, TimeSpan.FromSeconds(10),
        onBreak: (e, time) => _logger.LogWarning($"Request failed with {e.Message}. Pausing requests for {time}"),
        onHalfOpen: () => _logger.LogInformation("Enough time has passed, starting to accept requests again"),
        onReset: () => _logger.LogInformation("Successful request, resetting circuit"));

var response = await policy.ExecuteAsync(() => _client.SendAsync(request));
```

```
CircuitBreakerConfig circuitBreakerConfig = CircuitBreakerConfig.custom()
    .failureRateThreshold(50)
    .waitDurationInOpenState(Duration.ofMillis(1000))
    .permittedNumberOfCallsInHalfOpenState(2)
    .slidingWindowSize(2)
    .recordExceptions(IOException.class, TimeoutException.class)
    .ignoreExceptions(BusinessException.class, OtherBusinessException.class)
    .build();
```

CIRCUIT BREAKER
Einsatzzwecke

Hochschule RheinMain

- Temporale Probleme:
  - Gateway Timeout
  - Bad Gateway
  - Service Unavailable

- Ziel: Warten verhindern, schnelle Rückmeldung an den Nutzer, dass Subsystem derzeit nicht verfügbar ist

KOMBINATION VON MUSTERN

Hochschule RheinMain

- Warte und probiere nochmal, aber wenn zu viele Requests parallel Probleme haben, behandle den Service eine Zeit lang als unerreichbar

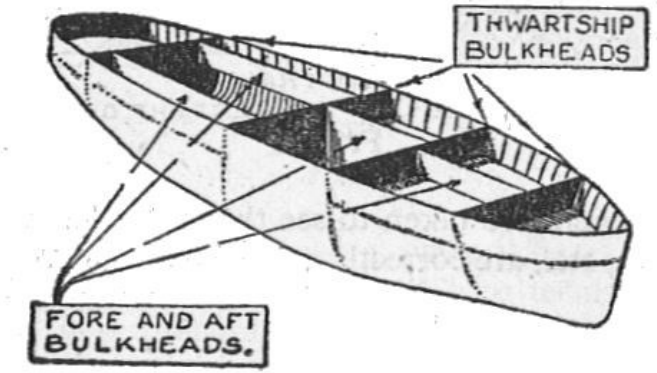
```
var exceptionsToHandle = Policy
    .Handle<HttpRequestException>(httpExc => httpExc.StatusCode == HttpStatusCode.InternalServerError);

var waitAndRetry = exceptionsToHandle
    .WaitAndRetryAsync(3, i => TimeSpan.FromSeconds(Math.Pow(2, i)),
        onRetry: (exc, waitTime) => _logger.LogWarning($"Request failed with {exc.Message}. Waiting {waitTime} to retry."));

var circuitBreaker = exceptionsToHandle
    .CircuitBreakerAsync(5, TimeSpan.FromMinutes(5),
        onBreak: (e, time) => _logger.LogWarning($"Request failed with {e.Message}. Pausing requests for {time}"),
        onHalfOpen: () => _logger.LogInformation("Enough time has passed, starting to accept requests again"),
    onReset: () => _logger.LogInformation("Successful request, resetting circuit"));

var policy = Policy.WrapAsync(circuitBreaker, waitAndRetry);

var response = await policy.ExecuteAsync(() => _client.SendAsync(request));
```

BULKHEAD

Hochschule RheinMain

- „Wegen einem Leck soll nicht gleich das ganze Schiff sinken“
- Begrenzung der Parallelität mittels Semaphor

```
var policy = Policy.BulkheadAsync(5,
    maxQueuingActions: 10,
    onBulkheadRejectedAsync: ctx =>
    {
        _logger.LogWarning("Bulkhead rejected request");
        return Task.CompletedTask;
    });

var response = await policy.ExecuteAsync(() => _client.SendAsync(request));
```



[Bild: Wikipedia]

```
BulkheadConfig config = BulkheadConfig.custom()
        .maxConcurrentCalls(10)
        .maxWaitDuration(Duration.ofMillis(1))
        .build();
```

WEITERE MUSTER

Hochschule RheinMain

- Timeout: Unbekannte Ausfälle erkennnen
- Fallback: Standardrückgabe im Fehlerfall
- Caching: Verhindern von Requests
- Steady State: Dauerzustand etablieren
- Fail Fast: Wartezeit auf Fehlermeldungen verhindern
- Test Harness: Tests mit fehlerhaften Protokollimplementierungen

Hochschule RheinMain

WIE HALTEN WIR EINEN CACHE DER DATENBANK AKTUELL?

Datenversionierung

THERE ARE ONLY TWO HARD THINGS IN COMPUTER SCIENCE: CACHE INVALIDATION, NAMING THINGS AND OFF-BY-1-ERRORS

Phil Karlton, Ergänzung von Leon Bambrick

DATENVERSIONIERUNG
Einführung

Hochschule RheinMain

- Problem: Replikation einer Datenbank, um Kommunikationsfehler auszugleichen
  - Datenbankinhalt des Replikats muss aktuell gehalten werden
  - Wie bewerkstelligen, ohne die gesamte Datenbank herunterladen zu müssen?

- Ansatz: Datenversionierung
  - Relationen enthalten explizites Versionsfeld (ordinal, e.g. numerisch oder Datum)
  - Replikation hält letzte synchronisierte Version
  - Jede Änderung inkrementiert immer die Version eines Datensatzes
    - Zähler über alle Datensätze einer Relation hinweg
  - Abfrage der seither geänderten Daten über einfache Abfrage
    - SELECT * FROM Tabelle WHERE Version > Threshold
  - Synchronisation über Bulk-Update

DATENVERSIONIERUNG
Beispiel

Hochschule RheinMain

SELECT * FROM Vorlesungen
Where Version > 3

Haupt-Datenbank

ID	Vorlesung	Professor	Version
1	Verteilte Systeme	Georg Hinkel	4
2	Softwaretechnik	Bodo Igler	3
…	…	…	…

Replikat

ID	Vorlesung	Professor	Version
1	Verteilte Systeme	Georg Hinkel	4
2	Softwaretechnik	Bodo Igler	3
…	…	…	…

ID	Vorlesung	Professor	Version
1	Verteilte Systeme	Georg Hinkel	4

DATENVERSIONIERUNG
Konkurrierende Zugriffe

Hochschule RheinMain

- Problem: Multi-Master-Replikation
  - Nebenläufige Änderungen an mehreren Replikaten
- Lösung: Zweistufiges Versionsfeld, ähnlich wie bei optimistischem Locking

ID	Vorlesung	Professor	Version	BasisV
1	Verteilte Systeme	Georg Hinkel	4	1
2	Softwaretechnik	Bernhard Turban	5	3
…	…	…	…	

ID	Vorlesung	Professor	Version	BasisV
2	Softwaretechnik	Bernhard Turban	5	1

ID	Vorlesung	Professor	Version	BasisV
2	Softwaretechnik	Bernhard Turban	5	3

DATENVERSIONIERUNG
Einsatzgebiete

Hochschule RheinMain

- Ausfallsicherheit
  - Zugriff auf Daten falls Konnektivität nicht gegeben
- Latenz
  - Replikat näher am Nutzenden kann Latenz reduzieren
- Partition
  - Replikat kann ggf. nur Teil der Daten vorhalten

Hochschule RheinMain

ABER WIE SIEHT ES MIT TRANSAKTIONEN AUS?

Verteilte Transaktionen

VERTEILTE TRANSAKTIONEN
Einführung

Hochschule **RheinMain**

- (Relationale) Datenbanken garantieren oft ACID-Eigenschaften
  - **A**tomizität: Transaktion wird entweder komplett oder gar nicht ausgeführt
  - (**C**)Konsistenz: System ist vor und nach einer Transaktion in einem konsistenten Zustand
  - **I**solation: Ignoriert weitere nebenläufige Transaktionen
  - **D**auerhaftigkeit: Wirkung einer Transaktion geht auch bei Ausfall nicht verloren

- Neues Problem: Einzelne Bestandteile einer Transaktion verteilt realisiert
  - Insbesondere bei Verwendung von Microservices
  - Dadurch ACID-Garantien eines DBMS nicht nutzbar
  - Stellenfehler (site failure) jederzeit möglich (Systemabsturz)

- Lösung: Commit-Protokolle

COMMIT-PROTOKOLLE

Hochschule RheinMain

- Dienen dazu, in verteilter Umgebung die Entscheidung Abort/Commit einer Menge von Prozessen zu koordinieren
  - Alle Prozesse, die Entscheidung treffen, treffen die gleiche Entscheidung
    ➔ Sobald ein Prozess Transaktion abbrechen will, müssen alle anderen auch abbrechen
  - Einmal getroffene Entscheidungen sind bindend für jeden Prozess, kann nicht widerrufen werden
  - Nach hinreichend langer Zeit kommt es zu einer Entscheidung

- Fenster der Verwundbarkeit
  - Unsicherheitsperiode zwischen lokaler Entscheidung und Kenntnis der Gesamtentscheidung
  - Man kann beweisen, dass man nicht ohne auskommen kann

- 2-Phasen-Commit-Protokoll
  - Am weitesten verbreitet

2-PHASEN-COMMIT-PROTOKOLL (2PC)
Einführung

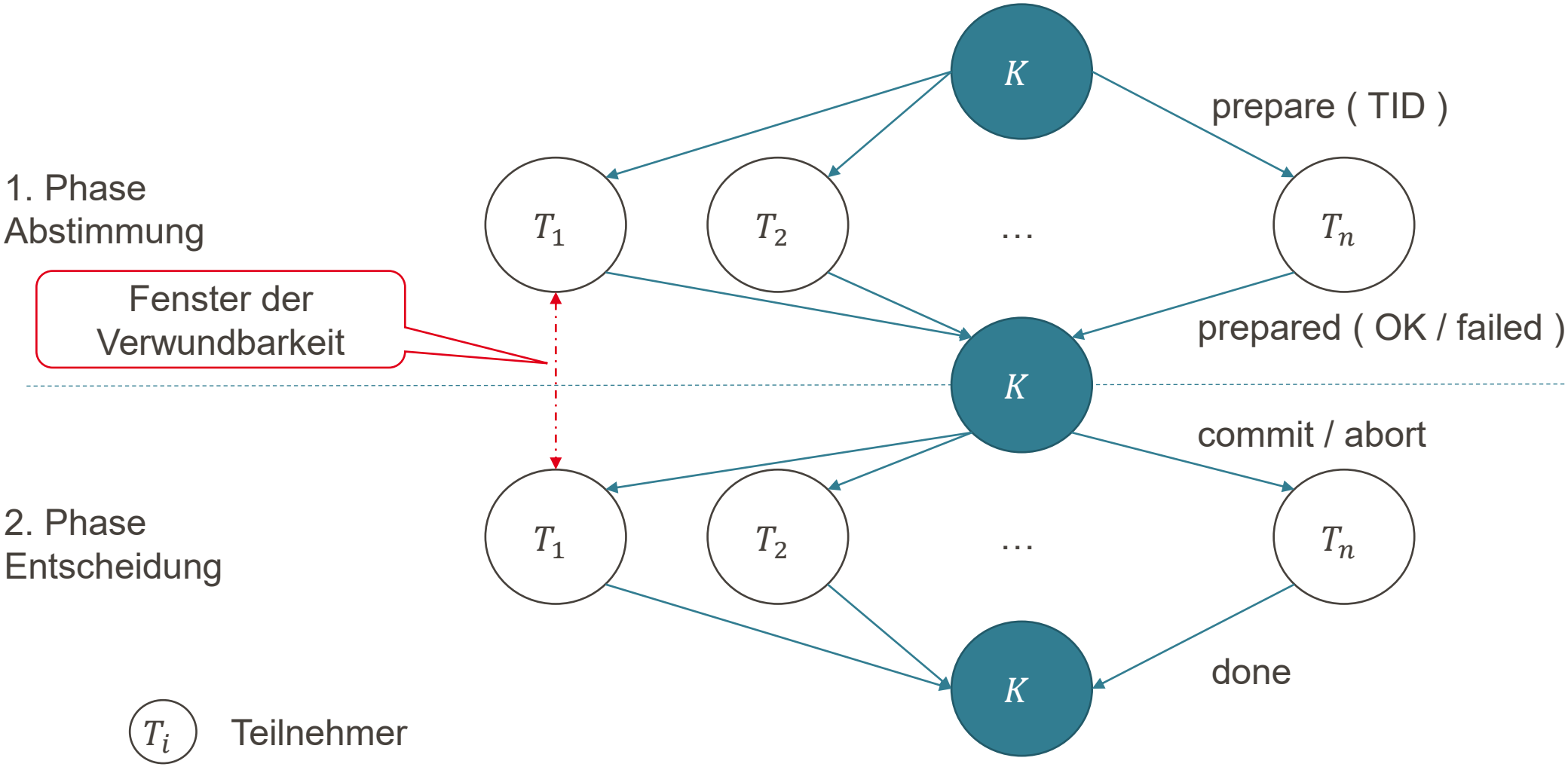
Hochschule RheinMain

- Blockierender Commit-Algorithmus mit schwacher Terminierungseigenschaft
  - Treten keine Fehler auf, treffen alle Prozesse irgendwann eine Entscheidung
  - Jeff Gray, 1978
- Zentraler Koordinator verwaltet Transaktionssteuerung, trifft Commit-Entscheidung
  - Idealerweise Deployment auf hochzuverlässigen Rechner
- Fehlerfall: Auswertung der jeweils lokalen Logs

2-PHASEN-COMMIT-PROTOKOLL (2PC)

Nachrichtenfluss ohne Fehler

Hochschule **RheinMain**



2-PHASEN-COMMIT-PROTOKOLL (2PC)
Fehlerfall

Hochschule RheinMain

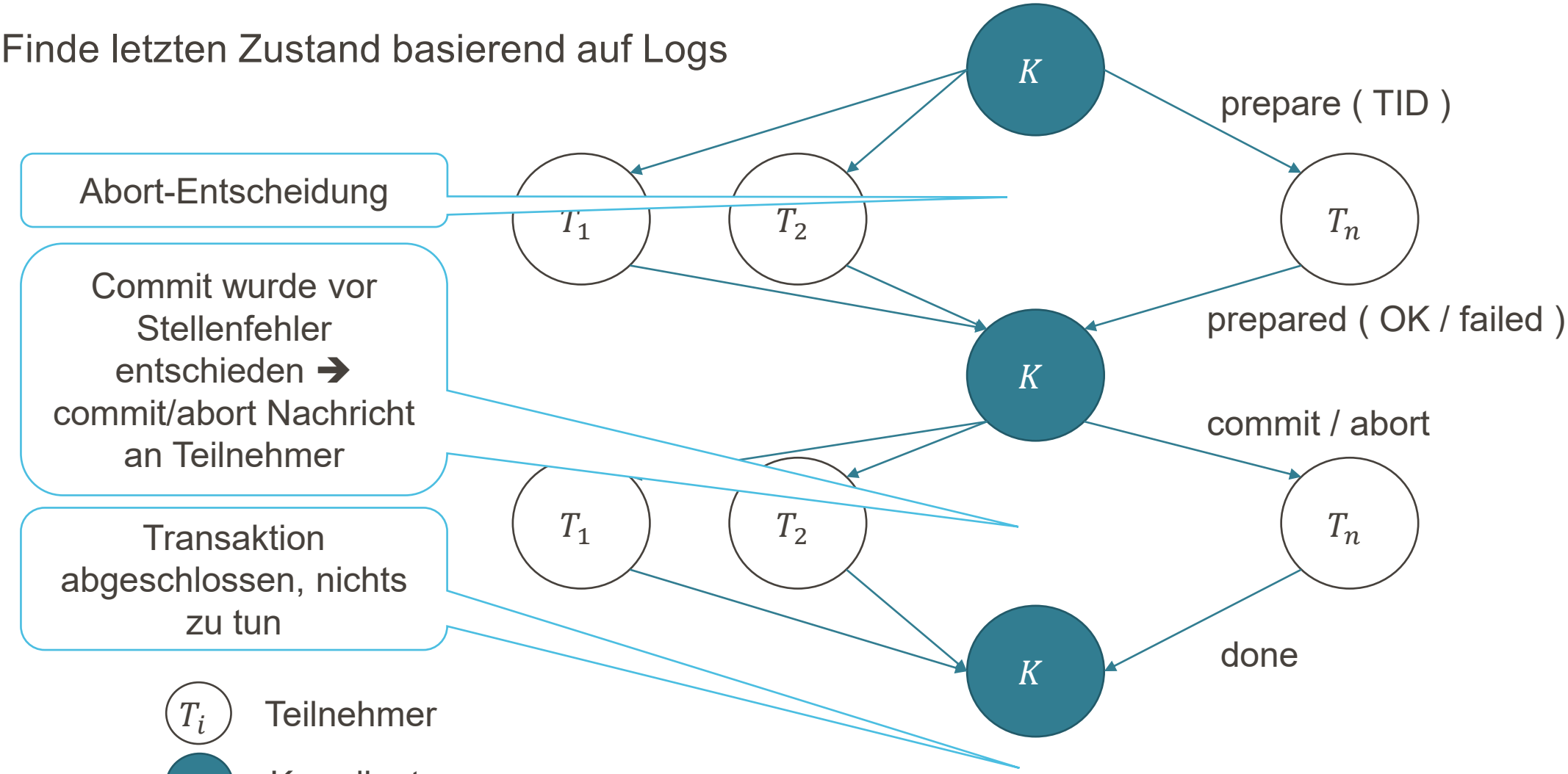
- Transaktionsfehler
  - Teilnehmer stellt selbst Fehler fest ➔ kommuniziert Fehler per prepared(failed) an Koordinator
  - Koordinator sendet Abort-Entscheidung aufgrund Transaktionsfehler bei einem Teilnehmer
- Kommunikationsfehler, Stellenfehler
  - Erkennung durch Timeouts
- Logging ermöglicht im Fall eines Stellenfehlers, Transaktion wieder aufzunehmen

2-PHASEN-COMMIT-PROTOKOLL (2PC)

Behandlung eines Stellenfehlers im Koordinator

Hochschule **RheinMain**

- Finde letzten Zustand basierend auf Logs

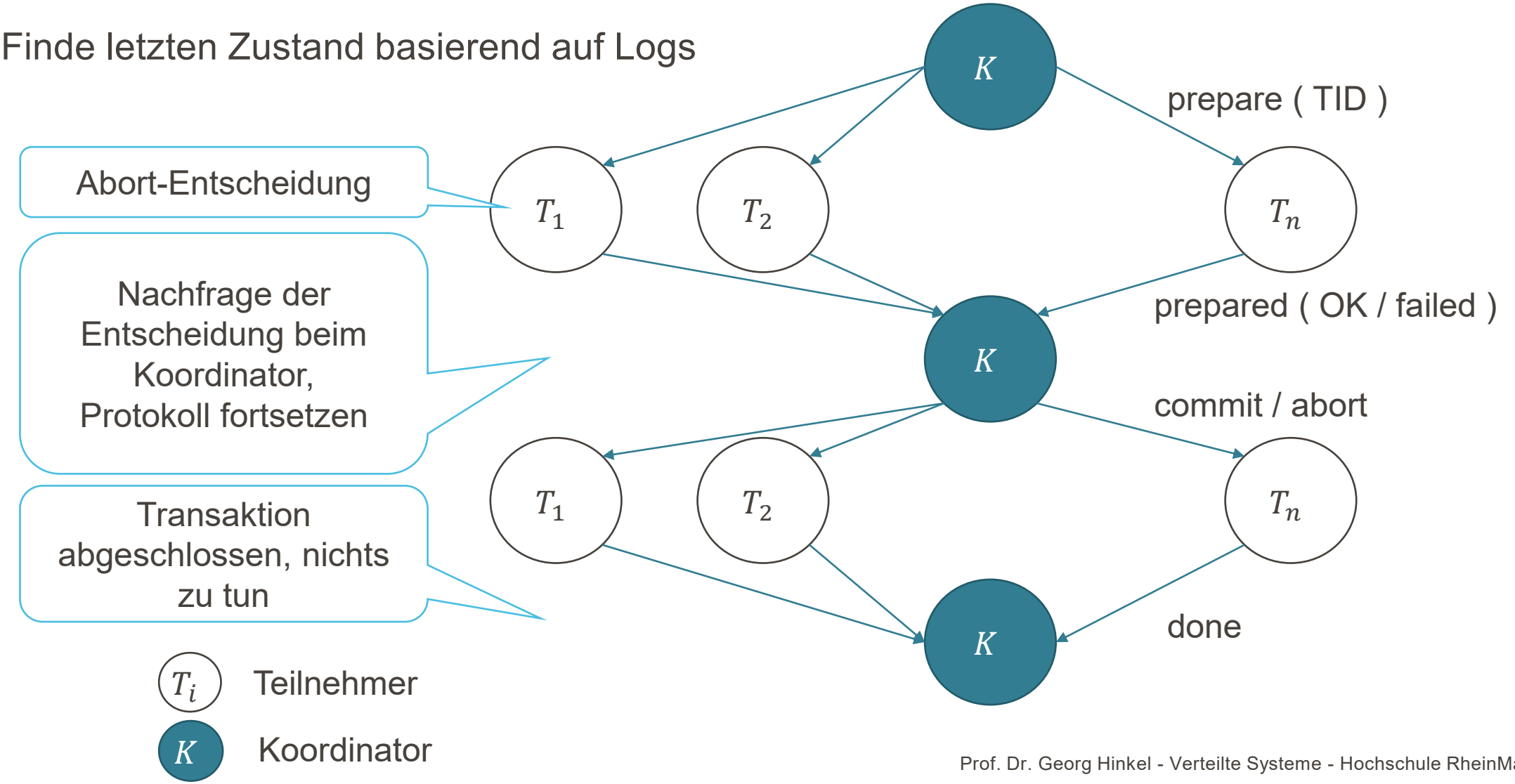


2-PHASEN-COMMIT-PROTOKOLL (2PC)

Behandlung eines Stellenfehlers im Teilnehmer

Hochschule **RheinMain**

- Finde letzten Zustand basierend auf Logs



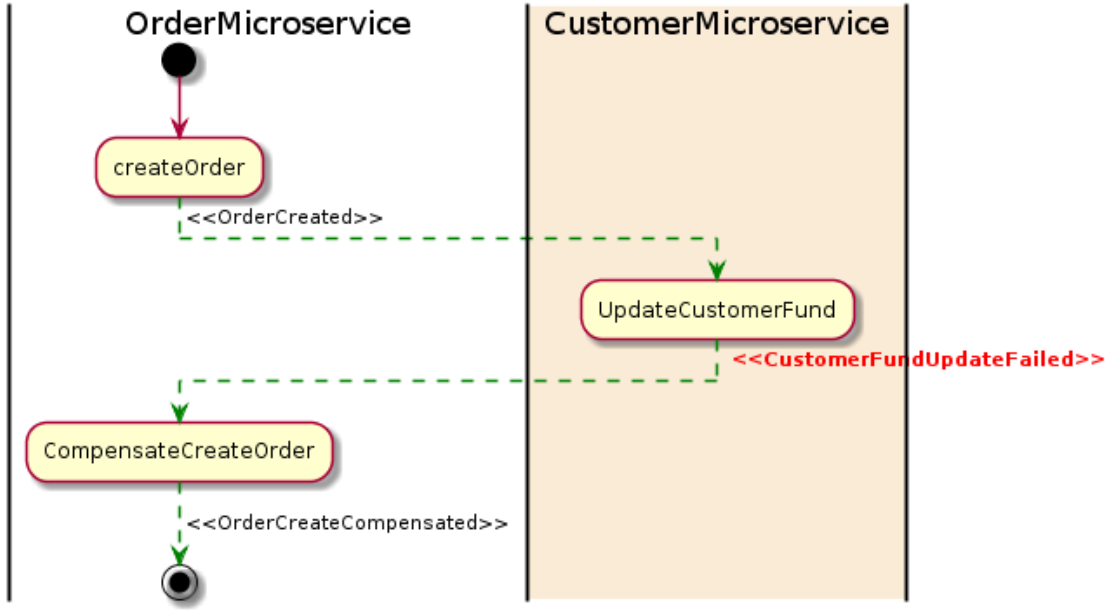
Hochschule RheinMain

UND WAS MACHEN WIR BEI FUNKTIONALEN ABHÄNGIGKEITEN ZWISCHEN TRANSAKTIONEN?

VERTEILTE TRANSAKTIONEN
Das Saga-Pattern

Hochschule RheinMain

- Idee: Führe einzelne Operationen individuell aus
  - Füge Undo-Operation einem lokalen Transaktionsspeicher hinzu
  - Typischerweise: Command-Pattern
- Im Fehlerfall: Kompensation der getätigten Operationen
  - Undo-Liste wird zurückgerollt



[https://developers.redhat.com/blog/2018/10/01/patterns-for-distributed-transactions-within-a-microservices-architecture]

ZUSAMMENFASSUNG

Hochschule RheinMain

- Muster für fehlertolerante/fehlertransparente Systeme
  - Retry
  - Retry and Wait
  - Circuit Breaker
  - Bulkhead

- Prinzipien für verteilte Datenhaltung
  - Datenversionierung zur Replikation von Daten
  - 2-Phasen-Commit für verteilte Transaktionen
  - Saga-Pattern für verteilte Transaktionen mit funktionalen Abhängigkeiten

MÖGLICHE PRÜFUNGSAUFGABEN

Hochschule RheinMain

- Was bedeutet Resilienz in verteilten Systemen?
- Welches Muster könnte man für eine gegebenes Szenario verwenden, um eine Fehlertoleranz zu erreichen?
- Wie funktioniert ein Circuit Breaker?
- Welche Änderungen müsste man an einem gegebenem relationalen Datenbankschema vornehmen, um Replikationen einfach synchronisieren zu können?
- Welche Änderungen müsste man an einem gegebenem relationalen Datenbankschema vornehmen, um gleichrangige Replikate der Datenbank untereinander synchronisieren zu können?
- Erläutern Sie das 2-Phasen-Commit-Protokoll!

ZUM WEITERLESEN

Hochschule RheinMain

- Michael Nygard: Release It!
  ISBN-13: 978-1680502398

- Martin Fowler: https://martinfowler.com/articles/microservices.html, https://martinfowler.com/bliki/CircuitBreaker.html
- Microsoft Docs: https://docs.microsoft.com/de-de/azure/architecture/patterns/