Grundlagen & Analyse verteilter Systeme

Modul 321

Kompetenzband A & B

Typische Eigenschaften verteilter Systeme

- **Latenz**: Zeit, die Daten zwischen Knoten benötigen
- **Konsistenz**: Gleichheit von Daten über alle Knoten
- **Nebenläufigkeit (Concurrency)**: Gleichzeitiger Zugriff auf Ressourcen
- **Fehlertoleranz**: Ausfallsicherheit bei Knoten- oder Netzwerkfehlern
- **Skalierbarkeit**: horizontal vs. vertikal

Latenz & Konsistenz (Visualisierung)

```
Client A ----> Server ----> Client B
200ms          200ms
```

- **Latenz**: 400ms bis B die Änderungen von A sieht
- **Konsistenz**: Strong vs. Eventual Consistency

Nebenläufigkeit

- Gleichzeitige Ausführung mehrerer Prozesse
- Beispiele:
  - Mehrere Clients greifen auf dieselbe Datenbank zu
  - Parallel verarbeitete Tasks in Microservices
- Herausforderung: Race Conditions & Deadlocks

Fehlertoleranz & Skalierbarkeit

- **Fehlertoleranz**: Replikation von Daten / Dienste redundant betreiben
- **Skalierbarkeit**:
  - **Horizontal**: zusätzliche Server
  - **Vertikal**: mehr Ressourcen auf einem Server

Architekturstile: Client/Server

- Zentraler Server, viele Clients
- Vorteile: einfache Struktur, zentrale Kontrolle
- Nachteile: Single Point of Failure
- Beispiel: Webserver + Browser

```
Client1   Client2
 \       /
  \     /
   Server
```

Architekturstile: Peer-to-Peer

- Gleichberechtigte Knoten, direkte Kommunikation
- Vorteile: Ausfallsicherheit, Lastverteilung
- Nachteile: Komplexe Konsistenz
- Beispiel: BitTorrent, Blockchain

```
Peer1 <-> Peer2 <-> Peer3
```

Architekturstile: Microservices

- Lose gekoppelte Services, eigene Datenhaltung
- APIs für Kommunikation
- Vorteile: unabhängige Deployment, Skalierbarkeit
- Nachteile: komplexes Monitoring & Netzwerk
- Beispiel: Online-Shop (Orders, Inventory, Payments)

```
[Orders] --> [Inventory] --> [Payments]
```

Analyse bestehender Architektur

- Prüfen von Kommunikationswegen
- Abhängigkeiten und Isolation identifizieren
- Skalierbarkeit, Nebenläufigkeit und Fehlertoleranz prüfen
- Eignung für Verteilung oder Monolith entscheiden

Datenhaltung in verteilten Systemen

- **Replikation**: Daten an mehreren Knoten → Ausfallsicherheit
- **Sharding**: Daten auf mehrere Knoten → Lastverteilung
- **Konsistenzmodelle**: Strong vs. Eventual
- **Transaktionen**: ACID vs. BASE
- **Backup & Recovery**: zentral oder inkrementell

Konsistenzmodelle

Modell	Beschreibung
Strong Consistency	Alle Knoten sehen sofort gleiche Daten
Eventual Consistency	Daten werden zeitverzögert konsistent
Weak Consistency	Keine Garantie über Datenkonsistenz

CAP-Theorem

- **C**onsistency: gleiche Daten an allen Knoten
- **A**vailability: System reagiert auf jede Anfrage
- **P**artition Tolerance: funktioniert trotz Netzwerkpartition

Trade-off: nur zwei von drei gleichzeitig möglich

```
+-------------+-------------+-------------+
| C           | A           | P           |
+-------------+-------------+-------------+
| Strong Cons | Eventual A  | Partition T |
+-------------+-------------+-------------+
```

Datenhaltung: Microservices

- Jeder Service eigene Datenbank
- Vorteile:
  - Isolation & Konsistenzkontrolle
  - Unabhängiges Deployment
- Herausforderung: Daten über Services synchron halten

Replikation & Sharding

- **Replikation**: gleiche Daten auf mehreren Knoten
- **Sharding**: Daten auf mehrere Knoten verteilen
- Vorteile: Ausfallsicherheit & Lastverteilung
- Beispiele:
  - MongoDB Sharding
  - PostgreSQL Replikation

Transaktionen in verteilten Systemen

- Typische Probleme: Deadlocks, Netzwerkfehler
- Modelle:
  - ACID: atomar, konsistent, isoliert, dauerhaft
  - BASE: Basically Available, Soft-state, Eventually consistent

Backup & Recovery

- **Vollbackup**: alle Daten sichern
- **Inkrementelles Backup**: nur geänderte Daten
- Anpassung an verteilte Replikation
- Ziel: schnelle Wiederherstellung bei Ausfall

Zusammenfassung

- Verteilte Systeme: Latenz, Konsistenz, Nebenläufigkeit
- Architekturen: Client/Server, P2P, Microservices
- Analyse: Kommunikationswege, Abhängigkeiten, Skalierbarkeit
- Datenhaltung: Replikation, Sharding, Konsistenz, Transaktionen, Backup
- CAP-Theorem: Trade-off zwischen Konsistenz, Verfügbarkeit und Partitionstoleranz