Probeklausur

Architektur von Anwendungssystemen

Universität:	Technische Universität Berlin
Kurs/Modul:	Architektur von Anwendungssystemen
Bearbeitungszeit:	120 Minuten
Erstellungsdatum:	September 19, 2025

Aufgabe 1.

(a) Beschreiben Sie den Schichtenarchitektur-Stil. Nennen Sie mindestens drei Vorteile und zwei Nachteile.

(b) Skizzieren Sie ein kurzes Architekturmodell eines Web-Frontends und eines Backend-Dienstes mit einer Datenbank. Nennen Sie drei zentrale Komponenten und deren Rollen.

(c) Diskutieren Sie, wie asynchrone Kommunikation in einer Architektur mit einer Messaging-Queue Latenz, Durchsatz und Fehlertoleranz beeinflusst. Nennen Sie drei typische Muster.

Aufgabe 2.

(a) Gegeben sei ein E-Commerce-System mit regional verteilter Infrastruktur und hohen Anforderungen an Verfügbarkeit. Welche Architekturformen und Middleware-Ebenen würden Sie bevorzugen und warum? Nennen Sie zwei konkrete Optionen und erläutern Sie kurz deren Nutzen.

(b) Beschreiben Sie die Unterschiede zwischen Service-Oriented Architecture und Microservices. Gehen Sie dabei auf Abgrenzung der Dienste, Governance und Skalierbarkeit ein.

(c) Welche Kriterien würden Sie bei der Auswahl von Middleware-Technologien berücksichtigen, um Verlässlichkeit, Skalierbarkeit und Sicherheit eines verteilten Systems sicherzustellen? Nennen Sie mindestens vier Kriterien und erläutern Sie deren Bedeutung.

Aufgabe 3.

(a) Entwerfen Sie eine Referenzarchitektur für eine verteilte Anwendung zur Datenverarbeitung in Echtzeit mit Streaming-Komponenten. Nennen Sie mindestens vier Komponenten und deren Rollen.

(b) Diskutieren Sie, wie Latenz, Durchsatz und Fehlertoleranz in der Wahl von Messaging-Technologien berücksichtigt werden sollten. Führen Sie drei relevante Überlegungen auf.

(c) Welche Rolle spielen Idempotenz und Deduplication in verteilten Systemen? Beschreiben Sie die Grundidee und einen typischen Einsatzfall.

Aufgabe 4.

(a) Skizzieren Sie eine Strategie zur Authentifizierung und Autorisierung in einer verteilten Architektur. Diskutieren Sie den Einsatz von Identity-Providern, JWTs und API-Gateways.

(b) Stellen Sie drei Strategien zur Gewährleistung der Datenkonsistenz in verteilten Systemen gegenüber. Erläutern Sie kurz CAP, eventual consistency und verteilte Transaktionen im Saga-Ansatz.

(c) Welche Maßnahmen erhöhen die Observability eines verteilten Systems? Gehen Sie kurz auf Logging, Tracing, Metrics und Alerting ein.

Lösungen

Aufgabe 1.

(a) Die Schichtenarchitektur (Layered Architecture) gliedert eine Anwendung in logisch getrennte Schichten, typischerweise Präsentation, Anwendungslogik, Domänenlogik und Persistenz. Jede Schicht hat klar definierte Verantwortlichkeiten und kommuniziert über standardisierte Schnittstellen mit der darüber bzw. darunterliegenden Schicht. Zusätzliche Querschnittsbelange wie Sicherheit, Logging oder Transaktionsmanagement werden oft horizontal über die Schichten hinweg adressiert.

Vorteile (mindestens drei):

- Klare Trennung der Verantwortlichkeiten erleichtert Wartung, Tests und Weiterentwicklung.
- Wiederverwendbarkeit von Komponenten in mehreren Schichten.
- Austauschbarkeit oder Ersetzung einzelner Schichten ohne komplette Neugestaltung des Systems.
- Ermöglicht parallele Entwicklung verschiedener Teams an unterschiedlichen Schichten.

Nachteile (mindestens zwei):

- Erhöhter Overhead und potenzielle Latenzen durch mehrere Hop-Folgen zwischen Schichten.
- Gefahr der Over-Architektur oder übermäßiger Abstraktionen, was Komplexität erhöht.

(b) Kurzes Architekturmodell eines Web-Frontends und eines Backend-Dienstes mit einer Datenbank. Drei zentrale Komponenten und deren Rollen:

- **Web-Frontend (UI/Client)** – Darstellung der Anwendung, Aufnahme von Nutzereingaben, Client-seitige Logik (z. B. JavaScript/TypeScript).
- **Backend-Dienst (API/Services)** – Geschäftslogik, Koordination von Transaktionen, Bereitstellung von REST/GraphQL-Endpunkten.
- **Datenbank** – Persistenz von Zustandsdaten, als relationales DBMS oder NoSQL-System; konsistente oder eventual konsistente Speicherung je nach Anforderungen.

Skizzierung (Textform): **Client** –HTTP–> **Backend-Dienst** –JDBC/ORM–> **Datenbank**. Optional kann zwischen Client und Backend ein API-Gateway/Load-Balancer liegen, um Lastverteilung und Authentifizierung zentral zu handhaben.

(c) Asynchrone Kommunikation in Architekturen mit Messaging-Queues beeinflusst Latenz, Durchsatz und Fehlertoleranz folgendermaßen:

- **Latenz**: Die End-to-End-Latenz kann durch das Messaging-Intermediär erhöht sein (Queueing, Persistenz im Broker, Bestätigungen). Allerdings sinkt die wahrgenommene Latenz oft beim Durchsatzmaximum durch Parallelisierung und Entkopplung.
- **Durchsatz**: Durch Entkopplung und horizontale Skalierung von Publishern/Consumer-Gruppen lässt sich der Durchsatz erhöhen. Partitionierung des Topics/Queues ermöglicht parallele Verarbeitung.

- **Fehlertoleranz**: Durable Queues/Topics, Wiederholungslogik, Dead-Letter-Queues (DLQ) und Retries verbessern die Fehlertoleranz. Desynchronisation zwischen Producer und Consumer wird dadurch abgefedert.

Typische Muster:

- **Publish-Subscribe** (Themen/Topics): Producer sendet Ereignis an ein Topic; mehrere Consumer erhalten dieselben Ereignisse.
- **Queue-based Point-to-Point** (Queues mit Bestätigung): Producer schreibt in eine Warteschlange; ein Consumer verarbeitet eine Nachricht; sorgt für Lastverteilung.
- **Event-Streaming / Log-basiertes Muster**: Ereignisse werden in ein Append-Only-Log geschrieben (z. B. Kafka-Themen); Consumeren in Ordung der Partitionen, oft mit Replay-Fähigkeit.

Aufgabe 2.

(a) Zwei konkrete Optionen zur Gewährleistung von Verfügbarkeit in einem E-Commerce-System mit regional verteilter Infrastruktur:

- **Option 1: Regionally distributed Active-Active Microservices mit regionalen Datenbanken**
  Nutzen: Geringere Latenz für regionale Nutzer, höhere Verfügbarkeit durch Mehrregionen, Ausfall einer Region führt nicht zum Totalausfall. Umsetzungsideen: je Region eigene Instanzen der Kern-Services, asynchrone Replikation der persistierten Daten (Multi-Region-Datenbanken), globaler Traffic-Manager oder DNS-basiertes Routing, ggf. gemeinsamen Cache-Stack über Regionen. Vorteile: schnelle Reaktionszeiten, höhere Resilienz; Nachteile: komplexere Konsistenzmodelle, Replikationslatenzen.
- **Option 2: Globales Active-Passive mit hot-Standby-Regionen (Disaster-Recovery)**
  Nutzen: einfache Konsistenzüberwachung, schnelle Failover in eine schwerer lastende Region, geringerer operativer Mehraufwand im Vergleich zu voll verteilten Active-Active-Systemen. Umsetzung: in der Primärregion stellen Sie eine standby-Region mit synchronisierter Infrastruktur bereit; im Failover-Fall wird der Verkehr dorthin umgeleitet; Replikation der kritischen Datenbanken erfolgt synchron oder asynchron je nach SLA. Vorteile: weniger Komplexität, klare Failover-Grenzen; Nachteile: potenziell längere migrate-times, ggf. Latenzspitzen während Failover.

Beide Optionen unterstützen hohe Verfügbarkeit und DR-Gehalt, unterscheiden sich aber in Komplexität, Konsistenzmodell und Betriebsaufwand.
(b) Unterschiede zwischen Service-Oriented Architecture (SOA) und Microservices:

- **Dienstgranularität**: SOA neigt zu groberen Diensten (Enterprise Services); Microservices setzen auf sehr feine, eigenständige Dienste, die jeweils eine begrenzte, klar definierte Geschäftsdomäne abbilden.
- **Gewährleistung von Datenownership**: Microservices besitzen typischerweise eigene, isolierte Datenbanken; in SOA teilen sich Dienste tendenziell eher eine zentrale oder gemeinsam genutzte Datenbasis.
- **Governance und Schnittstellen**: SOA setzt oft auf ein zentrales Governance-Modell und ein komplexes Messaging- bzw. ESB-Ökosystem; Microservices setzen auf dezentrale Governances, leichtgewichtige APIs, Standardprotokolle (REST/gRPC) und geringe zentrale Middleware.
- **Deployment und Unabhängigkeit**: Microservices sind in der Regel unabhängig deploybar, skalierbar und evolvierbar; SOA-Dienste können schwerer zu versionieren und langsamer zu aktualisieren sein, da sie stärker auf gemeinsam genutzte Infrastruktur angewiesen sind.
- **Architekturprinzipien**: Microservices folgen oft Domain-Driven Design (Bounded Contexts), lose Kopplung, Eventual Consistency, und orchestrieren über leichte Patterns; SOA fokussiert stärker auf Service-Abstraktionen über ein zentrales Integrationsschema (z. B. ESB).

(c) Kriterien bei der Auswahl von Middleware-Technologien zur Sicherstellung von Verlässlichkeit, Skalierbarkeit und Sicherheit:

- **Zuverlässigkeit und Konsistenzmodell**: Unterstützt das System robuste Fehlertoleranz (z. B. Persistenz, Replikation, Exactly-Once Semantics, Transaktionen oder sagabasierte Transaktionen)?
- **Leistung (Latenz/Throughput) und Skalierbarkeit**: Horizontal skalierbar, geringe Latenz, gute Partitionierbarkeit, ausreichende Backpressure-Handhabung.
- **Sicherheit**: Authentifizierung/Autorisierung, Verschlüsselung im Transit und at Rest, Rollen- und Berechtigungsmodelle, Auditbarkeit.
- **Interoperabilität und Ökosystem**: Unterstützung gängiger Protokolle (HTTP/REST, gRPC, AMQP, Kafka-APIs), Sprachenunterstützung, Verfügbarkeit von Tools, Monitoring, Clusterverwaltung.
- **Observability und Betrieb**: integrierte Monitoring-/Logging-Optionen, Metriken, Tracing, einfaches Troubleshooting, Support- und Dokumentationsqualität.
- **Verlässlichkeit des Betriebs**: Einfachheit der Bereitstellung, Aktualisierung, Rollbacks, Failover-Verhalten, Upgrades.

Aufgabe 3.

(a) Referenzarchitektur für eine verteilte Anwendung zur Echtzeit-Datenverarbeitung mit Streaming-Komponenten (mindestens vier Komponenten):

- **Datenquellen/Producer(s)** – Erzeugen Ereignisse aus Sensoren, Anwendungen oder Logs.
- **Message-Broker / Stream-Medium** – z. B. Apache Kafka als Append-Only-Log, das Ereignisse durchnumeriert speichert.
- **Stream-Verarbeitungs-Engine** – z. B. Apache Flink (oder Spark Streaming) zur statefulen Berechnung, Windowing, Joins und Transformationslogik.
- **State Store / Checkpointing** – Persistente State-Backends (RocksDB etc.) innerhalb der Streaming-Engine; sorgt für Exactly-Once Semantics und Fehlertoleranz.
- **Ausgabe-Speicherpunkte (Sink)** – Data Lake (S3/ADLS), Data Warehouse (BigQuery, Snowflake) oder Indizierung (Elasticsearch) zur weiteren Nutzung bzw. Reporting.
- **Metadaten Schema-Verwaltung** – Schema Registry (Avro/Protobuf) für konsistente Event-Formate; ggf. Governance.
- **Monitoring Observability** – Logging/Tracing/Metrics (OpenTelemetry, Prometheus, Grafana) zur Überwachung des Pipelines.

(b) Drei relevante Überlegungen bei der Wahl von Messaging-Technologien in Bezug auf Latenz, Durchsatz und Fehlertoleranz:

- **Latenz vs Durchsatz-Abwägung**: Synchrone Bestätigungen erhöhen Latenz, asynchrone Verarbeitung erhöht Durchsatz; Partitionierung und Consumer-Parallelität verbessern Durchsatz bei überschaubaren Latenz-Anforderungen.
- **Zuverlässigkeit und Semantik**: Wahl zwischen At-Least-One, At-Most-Once oder Exactly-Once; Persistenz, Wiederholungslogik, DLQ-Konzept und Transaktionsunterstützung beeinflussen Verhalten bei Fehlern.
- **Bestandteile der Ordnung und Konsistenz**: Ordnung innerhalb Partitionen ist oft gewährleistet, über Partitionen hinweg muss man Alternativen (z. B. deduplizierende Logik, Idempotenz-Schutz) berücksichtigen; Replaybarkeit (Historie) und Datenreplikation beeinflussen Konsistenzmodelle.

(c) Idempotenz und Deduplication in verteilten Systemen:

- **Idempotenz** bedeutet, dass wiederholte Anfragen oder Ereignisse das System deterministisch gleichen Zustand erzeugen, wie eine einzige Anfrage. Typischer Einsatz: Wiederholte Event-Verarbeitung oder API-Aufrufe, bei denen Netzwerkfehler auftreten können.
- **Deduplication** schützt vor doppelten Ereignissen, indem eindeutige IDs (Message IDs, Sequence Numbers) genutzt werden, um Duplikate zu erkennen und zu verhindern, dass sie Seiteneffekte auslösen.

Typischer Einsatzfall: Bestellverarbeitung, bei dem eine Bestellung durch mehrfache Nachrichten identisch verarbeitet werden soll. Setzen Sie bijektive-IDs in der Message oder im Event-Header, speichern Sie verarbeitete IDs in einem Deduplication-Store, und verwenden Sie Exactly-Once-Semantik where verfügbar (z. B. in Kafka-Transaktionen oder in der Anwendung durch Idempotenz-Checks).

Aufgabe 4.

(a) Strategische Lösung zur Authentifizierung und Autorisierung in einer verteilten Architektur:

- **Identity-Provider (IdP)**: Zentraler Identitätsanbieter, der Benutzer/Clients authentifiziert (OIDC, OAuth 2.0, OpenID Connect).
- **JWTs/Access Tokens**: Clients erhalten kurze, signierte Tokens mit Claims (Rollen, Berechtigungen) vom IdP; Tokens werden bei jeder Anfrage an API-Gateway/Backend vorgelegt.
- **API-Gateway**: Zentraler Einstiegspunkt, der Token-Signaturen validiert, Authentifizierung durchführt und ggf. JWT-Weitergabe an Backend-Dienste sicherstellt.
- **Rollenbasierte Autorisierung in Diensten**: Dienste prüfen Claims und erzwingen Access-Control-Policies; Microservices können zusätzliche Authorizer (Policy) verwenden.
- **Best Practices**: kurze Token-Lebensdauer, Token-Refresh-Mechanismen, Verwalten von Sperrfunktionen/Token-Revocation-Listen, TLS-Transportverschlüsselung, Auditing.

(b) Drei Strategien zur Gewährleistung der Datenkonsistenz in verteilten Systemen; Erläuterung von CAP, eventual consistency und Saga-Ansatz:

- **CAP-Theorem als Orientierung**: In verteilten Systemen muss man Nuggets treffen, ob Konsistenz (C), Verfügbarkeit (A) oder Partitionstoleranz (P) wichtiger ist. Abwägungen führen zu AP- oder CP-Modellen, je nach Anwendungsfall.
- **Eventual Consistency**: Systeme erlauben vorübergehende Inkonsistenzen, verfolgen eventualen Abgleich (z. B. asynchrone Replikation); Vorteile: hohe Verfügbarkeit und Skalierbarkeit; Nachteile: ggf. vorübergehende Inkonsistenzen in der Sicht der Clients.
- **Saga-Pattern (verteilte Transaktionen)**: Lang laufende Transaktionen werden in Teiltransaktionen zerlegt, die jeweils eigene Bestätigungen geben; bei Fehlern werden kompensierende Transaktionen ausgelöst, um den vergangenen Zustand wiederherzustellen.

(c) Maßnahmen zur Observability eines verteilten Systems (Logging, Tracing, Metrics, Alerting):

- **Logging**: Strukturierte Logs mit Correlation/Trace-IDs über alle Services; zentrale Log-Sammlung (ELK/EFK, Loki); konsistente Log-Formate und Log-Level-Skalierung; ausreichende Rotations- und Speicherstrategien.
- **Tracing**: Verteilte Traces (OpenTelemetry, Jaeger, Zipkin) zur Nachverfolgung von Requests über Microservices; propagierte Kontextinformationen ermöglichen Korrelationsanalyse und Latenz-Topologien.
- **Metrics**: Metriken auf System- und Anwendungsebene (Prometheus, Grafana); SLOs/SLIs definieren, Umgebungen vergleichen, Aggregationen für Fehlervorhersagen.
- **Alerting**: Schwellenwerte oder Anomalie-Erkennung; klare Alarmierungsregeln; Verbindung mit On-Call-Management; Verifikation von Alerts (False-Positive-Reduktion) und Eskalationspfade.