Lernzettel

Middleware-Konfiguration und -Betrieb: Messaging-Broker Konfiguration (RabbitMQ, Kafka), Service Registry, Load Balancing, Fault Tolerance

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

Lernzettel: Middleware-Konfiguration und -Betrieb: Messaging-Broker Konfiguration (RabbitMQ, Kafka), Service Registry, Load Balancing, Fault Tolerance

(1) Messaging-Broker-Konfiguration (RabbitMQ, Kafka). Messaging-Broker ermöglichen asynchrone Kommunikation zwischen Diensten. Zwei gängige Vertreter sind RabbitMQ und Kafka.

RabbitMQ – Kerndimensionen.

- Aufbau: Exchanges, Queues, Bindings. Messages gehen von einem Exchange zu passenden Queues.
- Routing-Modelle: Direct, Fanout, Topic, Headers.
- Sicherheit und Betriebsführung: TLS, Benutzer, vhosts, Policies, Queues mit Replikation (Mirrored oder Quorum Queues).
- Zuverlässigkeit: Publisher Confirms, Consumer Acknowledgments, Transaktionen (bzw. Publisher Confirms in asynchronem Modus).
- Typische Einsatzfälle: Befehl-basierte Workflows, Task-Queues, fanout-Verteilungen.

Kafka – Kerndimensionen.

- Aufbau: Topics, Partitions, Replicas, Brokers.
- Verteilung: Jeder Topic besteht aus Partitionen, die auf Broker verteilt sind.
- Konsum-Modell: Consumer Groups, Offset-Verfolgung, Skalierung durch mehrere Consumer.
- Speicherung: Append-only Logs, hohe Throughput, robuste Persistenz.
- Sicherheit: TLS, SASL, ACLs; Zentrale Konfigurationsparameter in server.properties bzw. KRaft-Konfiguration.

Grundkonfiguration – kurze Hinweise.

- RabbitMQ (rabbitmq.conf): Listener, Cluster, TLS, Virtual Hosts, Memory-/Queue-Quotas, Policy-basierte Policies.
- Kafka (server.properties): broker.id, listeners, advertised.listeners, log.dirs, num.partitions, default.replication.factor, retention.ms.

Typische Betriebsaspekte.

- Latenz vs. Durchsatz: RabbitMQ bevorzugt geringere Latenz bei zuverlässiger Zustellung; Kafka skaliert besser bei hohen Throughputs.
- Fehlertoleranz: Replikation, Quorum- bzw. ISR-Überwachung, Consumer-Offsets sauber verwalten.
- Observability: Metriken (Latenzen, Verarbeitungsgrad, Leerlaufzeit), Logs, Traces.

Beispiel-Konfigurationssnippet (lightweight).

- RabbitMQ: TLS-Verbindung, Direct-Exchange mit einer Queue pro Task-Type.
- Kafka: Topic mit Replikation Factor 3, min.insync.replicas = 2, retention.hours = 168.

Typische Muster.

- Async Command-Query-Pattern: Commands via Broker, Queries direkt zu Services.
- Event-Sourcing-/Event-Streaming-Muster: Ereignisse in Kafka persistent speichern, States aus Events rekonstruieren.

(2) Service Registry und Discovery. Service Registry dienen als zentraler Ort, an dem Dienste sich registrieren und Clients passende Endpunkte finden können. Sie unterstützen Health Checks und dynamische Endpunkt-Auflösung.

- Zweck und Nutzen: Entkopplung von Clients und Diensten, Lastverteilung, dynamische Skalierung.
- Typische Komponenten: Consul, Apache ZooKeeper, etcd, Eureka (Spring-Ökosystem).
- Funktionsweise: Dienste registrieren sich beim Start, regelmäßige Heartbeats/TTL, Clients fragen Registry nach Endpunkten.
- Health Checks: Pro-Instanz-Checks (HTTP, TCP, benutzerdefiniert) erhöhen Verfügbarkeit.
- Vorteile: Fehlersicherheit, skalierbare Architektur, bessere Beobachtbarkeit.

(3) Load Balancing. Load Balancing verteilt Anfragen auf mehrere Dienstinstanzen und sorgt für Verfügbarkeit.

- Client-seitig vs. serverseitig:
  - Client-side: Round-Robin, Weighted, Health Checks; z. B. Spring Cloud Load Balancer, Istio (seitens des Clients/Sidecar).
  - Server-seitig: NGINX, HAProxy, Envoy; TLS-Terminierung, health checks, Sticky Sessions.
- Typische Muster:
  - Only healthy instances werden belastet.
  - Zeitliche Lastverteilung durch Weighted-Strategien.
  - Satellite-Services hinter API-Gateway oder Ingress-Controller.

(4) Fault Tolerance. Ziel: Systemverfügbarkeit erhöhen, Ausfälle einschränken, keine Datenverluste bei Fehlern.

- Muster:
  - Retries mit Backoff (exponentiell), Timeout- und Circuit-Breaker-Logik.
  - Circuit Breakers (z. B. Resilience4j): Öffnen, wenn Fehlerhäufigkeit zu hoch, schließt wieder nach einer Haltezeit.
  - Bulkheads: Trennung von Ressourcen, um Ausfälle zu begrenzen.
  - Idempotenz und dedizierte Retry-Strategien, um Duplicate-Events zu vermeiden.
  - Backpressure-Strategien und Abbruchkriterien.

Semantische Überlegungen.

- Message Semantics: At-Least-Once, At-Most-Once, Exactly-Once-Garantie je nach Broker und Szenario.
- Konsistenz vs. Verfügbarkeit: Trade-offs in verteilten Architekturen.

(5) Praxis-Hinweise und Architekturüberblick.

- Observability: End-to-End-Tracing (OpenTelemetry), Metriken (Prometheus), Logging.
- Sicherheit: TLS, Auth, Autorisierung, Secrets-Management.
- Skalierung: Trennung von Daten- und Interaktionspfaden; mehrinstanzen, Sharding von Topics/Queues.
- Migration/Upgrade: Rolling Updates, Canary Deployments, Feature Flags.

Beispiel-Architekturtext (beschreibung).

- Ein Client sendet Anfragen über einen Load Balancer an Service-Instanzen hinter einem Registry-Eintrag.
- Dienste registrieren sich im Registry, Health Checks sichern Verfügbarkeit.
- Interne Kommunikation nutzt RabbitMQ oder Kafka je nach Muster (Befehle asynchron, Events streaming).
- Failover-Szenarien nutzen Circuit Breaker und redundante Instanzen, um Ausfälle zu überbrücken.