Verteilte Systeme

...für C++ Programmierer

Systemarchitektur

by

Dr. Günter Kolousek

Systemarchitektur – Definition

> *Die Architektur regelt den konzeptionellen Zusammenhang zwischen den verschiedenen* ***eigenständigen Komponenten*** *eines Systems. Sie formt die* ***logische*** *und* ***physikalische*** *Struktur eines Systems mit allen strategischen und taktischen* ***Entwurfsentscheidungen****, welche während dem Entwicklungsprozess angewendet werden müssen.*
>
> Grady Booch

Pragmatische Variante: Man versteht darunter die Konzipierung des Aufbaus von IT-Systemen aus Hardware, System-, Middleware- und Anwendungssoftware, Netzwerkstrukturen, Betriebspersonal und Nutzern.

Systemarchitektur – Kriterien

Allgemeine Kriterien für eine gute Architektur

- Einfachheit
  - komplex vs. kompliziert, $\rightarrow$ KISS
- Erweiterbarkeit: Änderung des Problemraumes
- Skalierbarkeit: Änderungen des Problemumfangs
- Kapselung: $\rightarrow$ Austauschbarkeit
- Sicherheit

Überblick über Architekturmuster

- Einteilung nach der Struktur
- Einteilung nach dem Informationsfluss
- Weitere Patterns

Einteilung nach der Struktur

grundlegende Architekturpatterns

- Schichtenarchitektur
- Peer-to-peer
- Pipe and Filter
- Broker
- Blackboard

Schichtenarchitektur

- engl. layered architecture
- Abstraktion: abhängig von Sicht (engl. view) werden wesentliche Merkmale (engl. feature) extrahiert $\rightarrow$ Schicht (engl. layer)
- Hierarchie von Abstraktionen: geordnete Reihenfolge
  - Zerlegung (engl. decomposition) eines Problems in Teilprobleme
  - Schnittstelle zwischen Schichten (engl. tiers)
  - Kapselung (engl. encapsulation): innere Struktur einer Schicht nicht einsehbar
- strikt vs. nicht strikt
- HW vs. SW

Schichtenarchitektur – 2

- Client/Server
- Proxy
- Load Balancer
- Result Cache
- Scatter und Gather

Client/Server

- Teile: Client, Server, Service
- Schichtenaufbau
  - Präsentationsebene (presentation layer, PL)
    - Darstellung der Daten, Verarbeitung von Benutzereingaben
  - Verarbeitungsebene (application layer, AL)
    - Business-Logic: Verarbeitung und Auswertung der Daten
  - Datenebene (data layer, DL)
    - Daten: persistent, unabhängig von AL, konsistent

Client/Server – Architekturen

- 2-Tier
  - Client: PL — Server: PL, AL, DL
  - Client: PL — Server: AL, DL
  - Client: PL, AL — Server: AL, DL
  - Client: PL, AL — Server: DL
  - Client: PL, AL, DL — Server: DL
- 3-Tier
  - Client: PL
  - Anwendungsserver: AL
  - Datenserver: DL
- n-Tier
  - Web
  - Steuerungsschicht zum Ablauf mehrerer fachlich abgegrenzter Teile des AL

Client/Server – Kriterien

- Antwortzeitverhalten, Netzwerkbelastung (Durchsatz)
  - abhängig von Netzwerk, Server, Client, SW,…
- Zuverlässigkeit und Verfügbarkeit
  - Redundanz der Daten durch Replikation
  - Redundanz der HW (Server, Netzwerk)
  - Redundanz der SW
  - HW-Tausch,…
- Skalierbarkeit
  - vertikal: weitere Ressourcen wie Speicher, schnellerer Prozessor,…
  - horizontal: weitere Ressourcen wie zusätzliche Hosts $\rightarrow$ Anpassung der SW
- Sicherheit

Client/Server – Kriterien – 2

- Integration in Legacy-Systeme
  - Import & Austausch von Daten, Authentifizierung, Zugriffskontrolle,…
- Flexibilität
  - Austausch der Benutzerschnittstelle,…
- Installation und Wartbarkeit
  - Installation? Warten? Deployment?
- Administration
  - z.B. Benutzerverwaltung
- Programmierproduktivität
  - abhängig von Anforderungen, gewählter Systemarchitektur und Tools
- Kosten: Lizenzen, Eigentumsverhältnisse,…

Proxy – Zweck und Struktur

Zweck

- Stellvertreter für Server
  - gleiches Interface
- fügt Funktionalität zum angeforderten Dienst
  - Performance: Caching, Lastverteilung
  - Filtern der Daten (auch Anonymisierung)
  - Zugriffskontrolle (auch Bandbreitenkontrolle)
  - Loggen der Zugriffe

Struktur

Proxy – Varianten, Vor- und Nachteile

Varianten

- transparenter Proxy
- Reverse Proxy

Vorteile

- Schutz der Clients und Server

Nachteile

- Gefahr durch Falschkonfiguration
- weitere Indirektion $\rightarrow$ Performance

Load Balancer

- ein Reverse-Proxy
- Dispatcher (reverse-proxy) entscheidet auf Basis von
  - Zufall
  - Round robin
  - geringste Auslastung
  - Session (z.B. Cookie)
  - Parameter in Request

Result Cache

- ein Reverse-Proxy
- Ablauf
  1. im Cache nachschlagen
  2. wenn gefunden, dann zurückliefern
  3. andererseits zum Worker weiterleiten und Ergebnis in Cache

Scatter and Gather

”zerstreuen und einsammeln”

Zweck

- Aufteilung einer Anforderung auf viele Worker
  - bestes Ergebnis
  - Lastaufteilung
  - Redundanz

Aufbau

- Client
- Dispatcher
- mehrere Worker

Ablauf

1. Broadcast der Anforderung an alle Worker
2. auf alle Antworten warten
3. Einzelantworten zu Gesamtantwort zusammenfassen

Peer-to-Peer

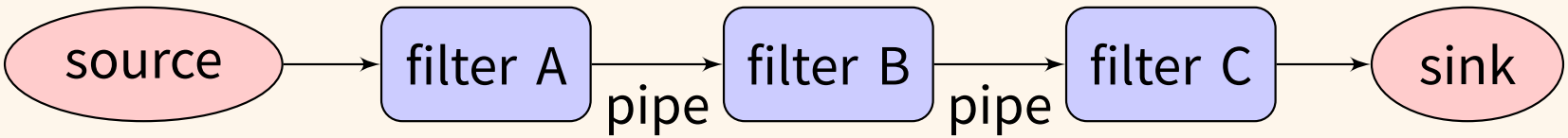
- Client/Server: Server Flaschenhals?!
- keine Server mehr!
- d.h. Client und Server-Rollen wechseln nach Bedarf
- Nachteile
  - einheitlichen Status der Applikation bestimmen/gewährleisten
  - effizientes Routing muss sichergestellt werden können
  - Finden eines Kommunikationspartners
  - Netzwerkausfälle maskieren
- Beispiele: file sharing (z.B. BitTorrent), Blockchain (z.B. Bitcoin), anonyme Internetbenützung (z.B. I2P)

Pipe and Filter – Zweck und Struktur

Zweck

- Data-Flowarchitektur
- inkrementelle Transformation der Daten in jedem Verarbeitungsschritt

Struktur



- Pipe $\rightarrow$ FIFO
- Filter: teilen sich keinen Zustand, haben kein Wissen über andere Filter

Subtypen

- Pipelines: lineare Topologie
- Bounded Pipes: Menge der Daten ist begrenzt
- Getypte Pipes: Daten haben Typ

Pipe and Filter – Vorteile

- keine komplexen Interaktionen, Filter (Worker) sind Black Boxes
- leichte Zusammensetzbarkeit, hierarchische Strukturierung möglich
- leichte Wiederverwendbarkeit
- parallelle Verarbeitung möglich
  - d.h. mehrere parallele Filter (Worker) je Stufe
- keine Zwischenspeicher (wie Dateien,…) notwendig
  - Implemenetierung als Queue
- schnelles Prototyping

Pipe and Filter – Nachteile

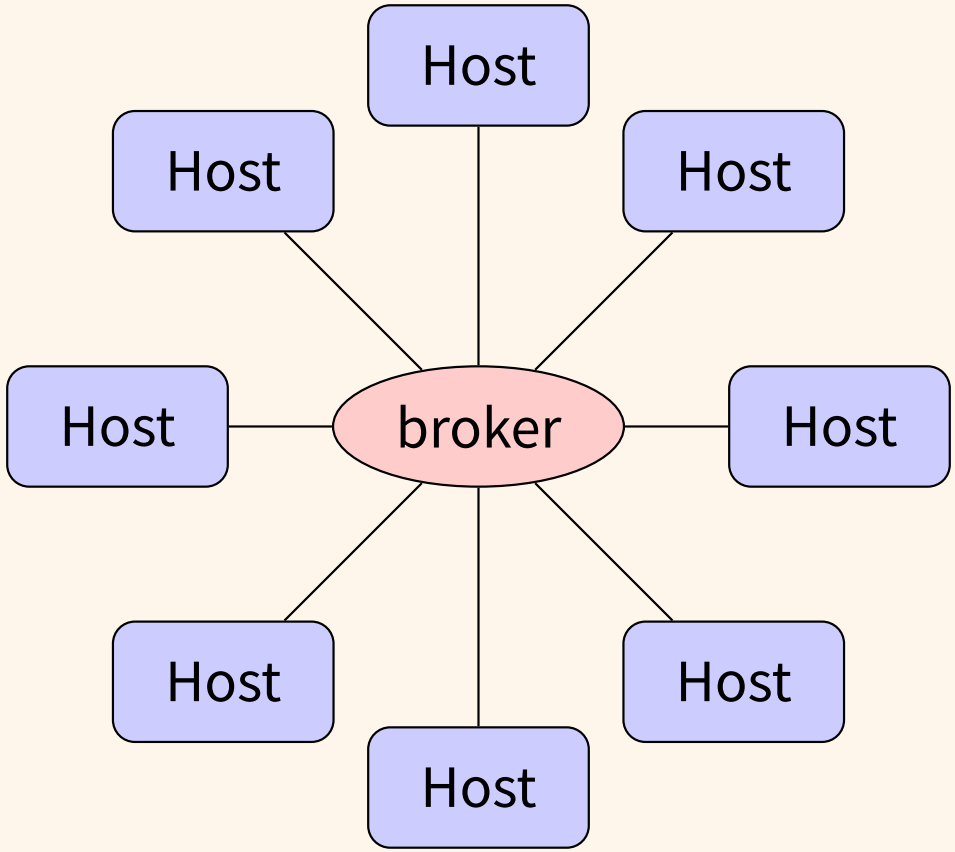
- Fehlertoleranz (was ist wenn Filter ausfällt?)
- Performance (jeder Filter muss Daten parsen, keine globalen Daten)
- nicht direkt für interaktive Anwendungen
- Filter können nicht gemeinsam an einem Problem arbeiten
- Pufferkapazität der Filter muss prinzipiell unbegrenzt sein

Broker

Zweck

- Entkoppeln von Sender und Empfänger, sodass Kommunikation möglich ist: Broker empfängt, bestimmt Ziel, leitet Nachricht weiter!

Struktur



Blackboard

aka Shared Space, auch Tuple Space

- siehe Foliensatz ”Serverprogrammierung”
- Struktur
  - Client: stellt Request in Space
  - Blackboard
  - Worker: arbeiten Requests ab
- Ablauf eines Workers
  1. aktuelles Zwischenergebnis vom Blackboard holen
  2. neuen Wert zum Zwischenergebnis hinzufügen
  3. neues Zwischenergebnis im Blackboard ablegen

Einteilung nach Informationsfluss

- Pull-Architektur: Client 'pulled' vom Server
  - Client muss wissen **wo** und **wann** Informationen verfügbar sind $\rightarrow$ regelmäßig abfragen $\rightarrow$ Request/reply Messaging Pattern
- Push-Architektur: Server 'pushed' zum Client
  - große Mengen von Information an viele Clients
  - Anbieter klassifiziert Information $\rightarrow$ Kanal, Interessenten abonnieren Kanal
  - Beispiele: E-Mail, Usenet News
- Event-Architektur:
  - ähnlich Push, aber kleinere Informationseinheiten, Subskription basierend auf Event-Klassen, Event-Mustern, bestimmten Events
  - problematisches Event-Routing, Ressourcenverbrauch

Weitere Patterns

- weitere Architekturpatterns
  - MapReduce
  - SOA
  - ESB
  - MOM
- SW und Architekturpattern
  - Ports and Adapters
- SW
  - Middleware

MapReduce

- Hauptkomponenten (bei Google, 2008!)
  - GFS (Google File System): verteiltes Dateisystem
    - mehr als 200 Cluster, die auf GFS basieren!
    - einige Installationen bei Google: ”many petabytes in size”
  - Bigtable: ”NoSQL Big Data database service”
  - *MapReduce*: Programmiermodell und Implementierung zum verteilten Berechnen
    - 100000 MapReduce Jobs jeden Tag: jeder benötigt 400 Server und ca. 5-10 Minuten!
  - fehlertolerante SW
    - fällt ein Server bei GFS, Bigtable oder MapReduce aus…

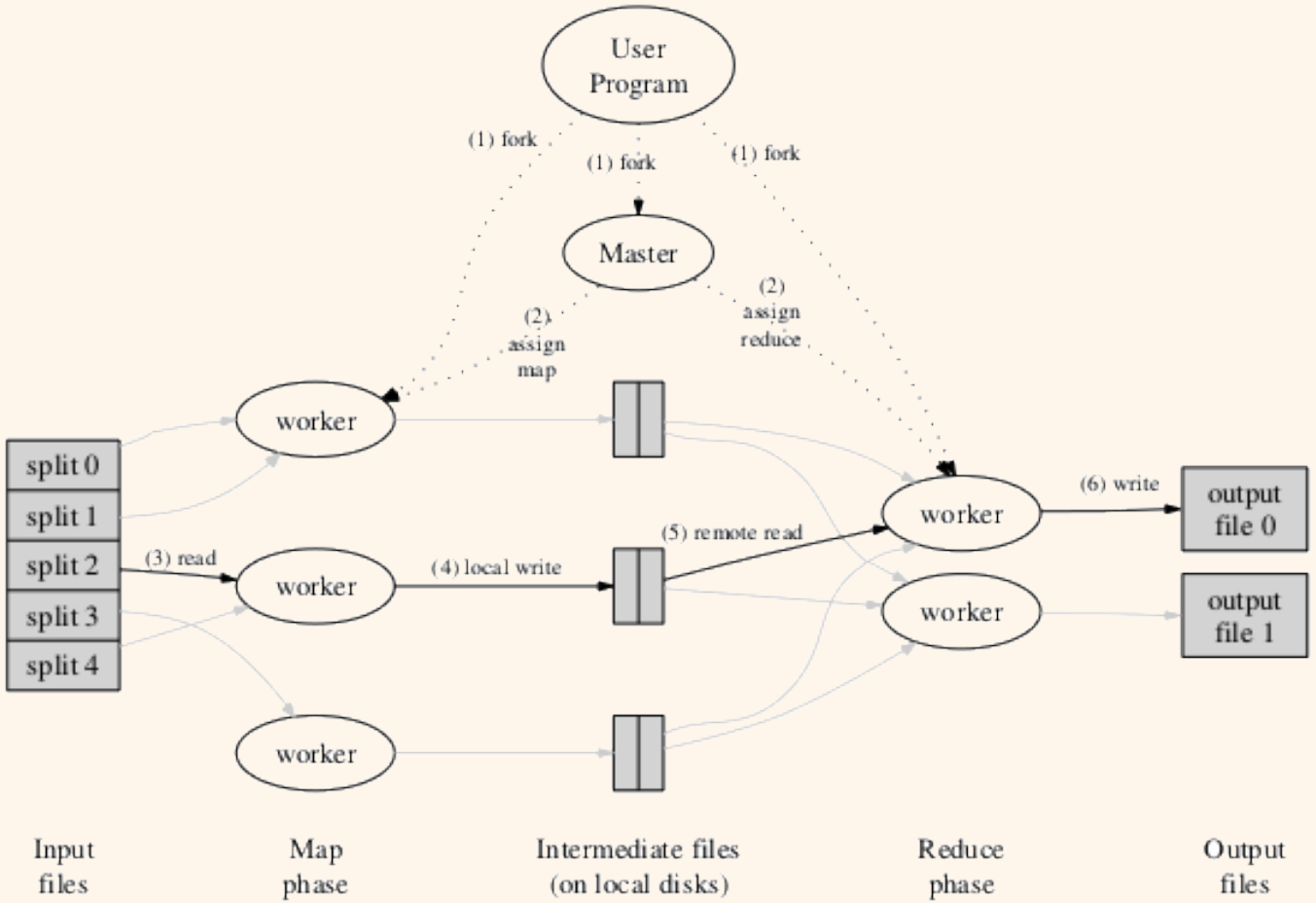
MapReduce – 2

- MapReduce bei Google
  - https://research.google.com/archive/mapreduce.html
  - Map/Reduce: Anleihen bei funktionaler Programmierung
    - $\rightarrow$ Open Source: Apache Hadoop
  - 2014: Cloud Dataflow: zusätzlich (zu batch mode) ”streaming data processing”
    - $\rightarrow$ Open Source: Apache Beam
- Anwendungen
  - invertierte Indizes, Graphstrukturen von Webdokumenten, machine learning, sortieren, einfache statistische Berechnungen
    - inverted index: Abbildung von Inhalt zu Ort, z.B. Wort $\rightarrow$ Dokumente (u.U. mit Position)

MapReduce – 3

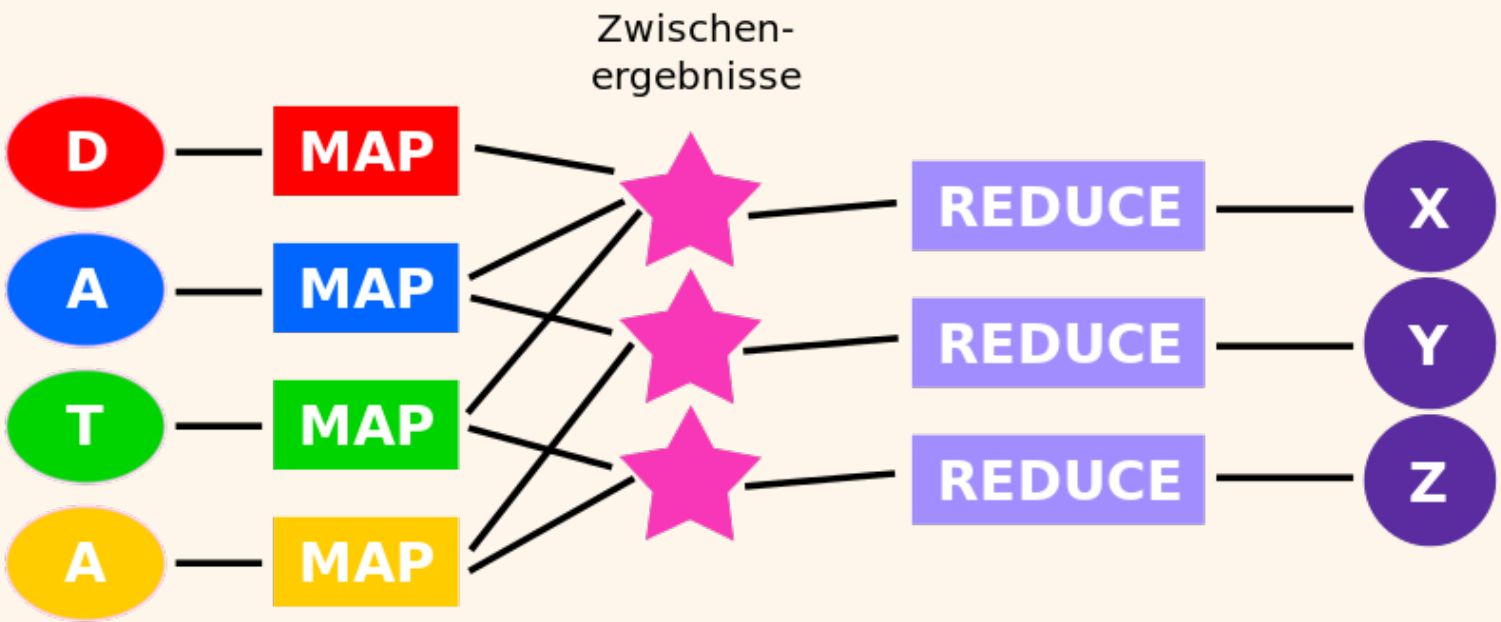
- Phasen
  1. Map: Daten aus GFS
     - Funktion Map wird vom Benutzer vorgegeben
  2. Combine (optional)
     - arbeitet wie Reduce, aber am Knoten von Map $\rightarrow$ reduziert die Datenmenge und somit die Netzwerkbelastung!
  3. Shuffle: Zuordnung der Ausgangsdaten der Map-Prozesse auf Eingangsdaten der Reduce-Prozesse
     - Teil des Frameworks
  4. Reduce: Daten in GFS
     - Funktion Reduce wird vom Benutzer vorgegeben

MapReduce – 4



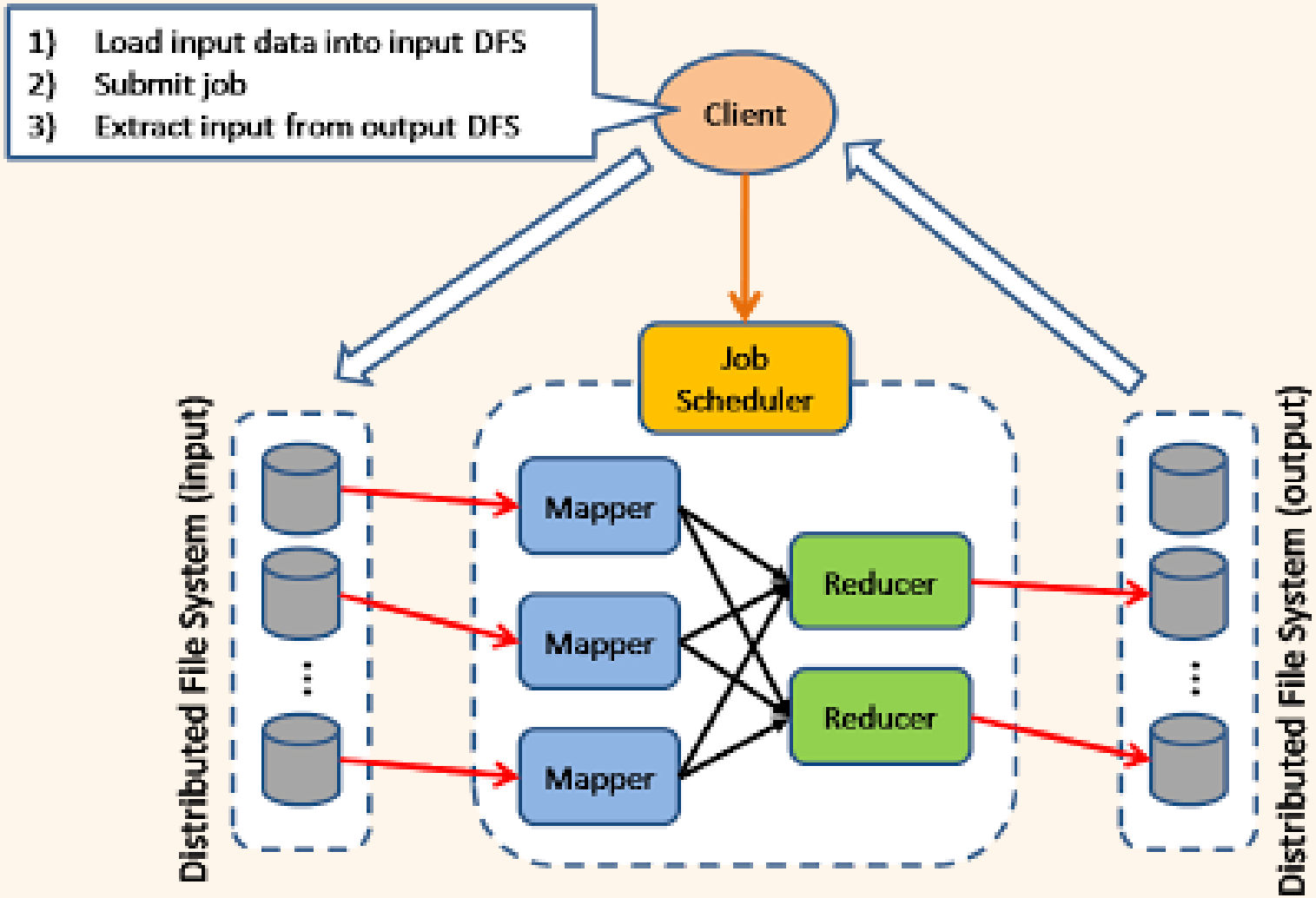
Quelle: https://research.google.com/archive/mapreduce.html (2004)

MapReduce – 5



Quelle: https://de.wikipedia.org/wiki/MapReduce

MapReduce – 6



Quelle: http://horicky.blogspot.co.at/2010/10/scalable-system-design-patterns.html

MapReduce – 7

Zählen der Vorkommen der Wörter in einer größen Ansammlung von Wörtern:

```
map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, ”1”);
        // ko: or whatever the count will be

reduce(String key, Iterator values):
    // key: word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(AsString(result))
```

Quelle: https://research.google.com/archive/mapreduce.html (2004)

Service-oriented Architecture (SOA)

Service Oriented Architecture (SOA) is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains.

Reference Model for Service Oriented Architecture 1.0

A service is a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description.

Reference Model for Service Oriented Architecture 1.0

Service-oriented Architecture – 2

- Ein Dienst…
  - repräsentiert eine fachliche Funktionalität
  - ist in sich abgeschlossen und kann eigenständig benutzt werden
  - ist über das Netzwerk verfügbar
  - hat eine wohldefinierte Schnittstelle
    - Black-Box: Implementierung in beliebiger Programmiersprache
- Eine ”Anwendung” $\rightarrow$ Koordination der Dienste (Orchestrierung)

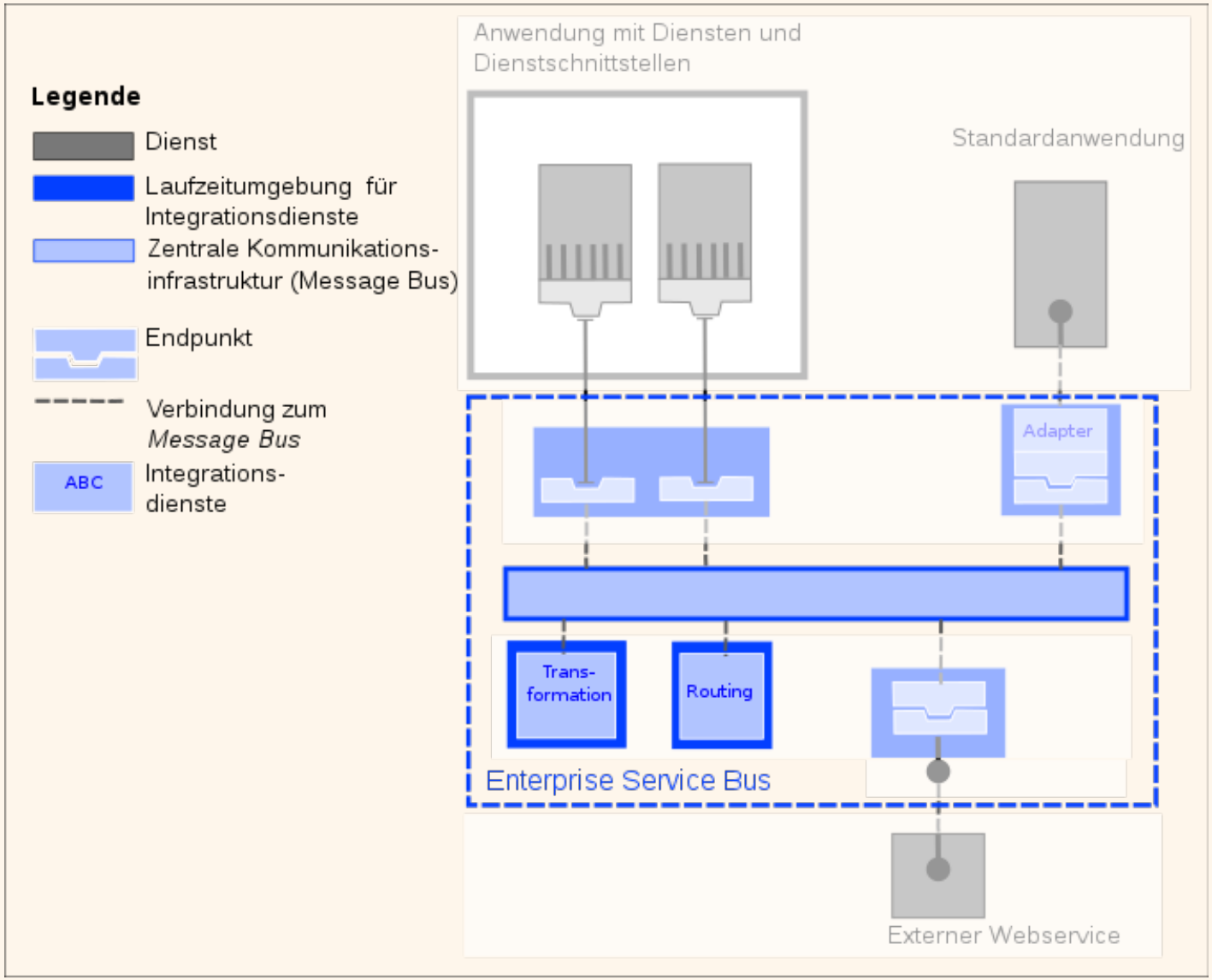
Service-oriented Architecture – 3

- Autovermietung
  1. Benutzer registrieren
  2. Reservierung vornehmen
  3. Mietvertrag erstellen
  4. Auto aushändigen
  5. Auto zurückgeben
  6. Abrechnung erstellen
- $\rightarrow$ jeweils ein Dienst
  - Nutzung jeweils auch für andere Geschäftsfälle
    - z.B. *Benutzer registrieren* für Autoversicherung

Service-oriented Architecture – 4

- Verbindung der Dienste mittels:
  - nachrichtenbasierter Kommunikation
    - meist: Punkt-zu-Punkt – Verbindungen
    - Serialisierung: ASN.1, YAML, JSON, BSON, MessagePack, Google Protobuf, Thrift,…
  - $\rightarrow$ Message Oriented Middleware
  - CORBA, ICE, Java RMI, WCF, grpc,…
  - Web Services basierend auf WS-* Spezifikationen (SOAP, WSDL, UDDI, WS-Security,…, XML-RPC) bzw. auf $\rightarrow$ REST
  - Enterprise Service Bus (ESB)
    - **Datenbus**, um Dienste in einem Unternehmensnetzwerk zur Verfügung zu stellen
    - Dienste sind über Endpunkte mit Bus verbunden
    - Austausch von Nachrichten um Dienste in Anspruch zu nehmen
    - z.B. IBM WebSphere ESB, MS BizTalk Server, Mule ESB, Apache ServiceMix
- Spezialfall: Microservices

ESB



Quelle: Wikipedia

Message Oriented Middleware

Zweck

- Abstraktion einer persistenten nachrichtenorientierten Kommunikation
- Nachrichten auf höheren Abstraktionsebene (nicht auf Bit/Byteebene)
- validieren, transformieren, weiterleiten

Definition: MOM ist

- eine Softwareinfrastruktur, die
- durch asynchrone Verbindungen charakterisiert ist und
- mehrere Systeme durch
- Nachrichten miteinander verbindet

Message Oriented Middleware – 2

Aufbau

- Broker: einer oder mehrere
- Server: registrieren sich bei Broker
- Clients: sendet Nachricht über Broker

Kommunikationsmodelle

- Message Queueing
- Publish/subscribe

Dienste

- Transaktionen, Prioritäten, Filterung, Transformation

Message Oriented Middleware – 3

QoS

- Zuverlässigkeit, Priorität, Time-to-Live

Sicherheit

- Authentifizierung, Geheimhaltung, Integrität, Zugriffskontrolle

Message Oriented Middleware – 4

Vorteile

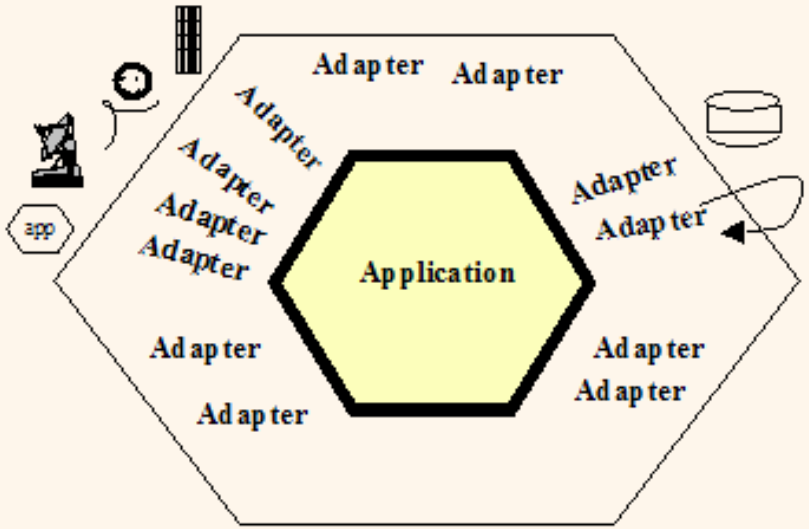
- Lose Kopplung
- asynchrone (und synchrone) Kommunikation
  - Server muss nicht online sein!
- Lastverteilung und parallele Verarbeitung möglich
- Verfügbarkeit einzelner Teilsysteme

Nachteile

- MOM ist SPOF (single point of failure)
- Routing der Nachrichten

Ports and Adapters

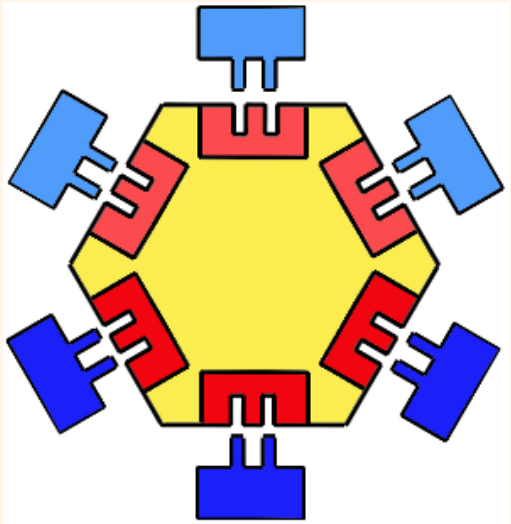
- alternativer Name: Hexagonal Architecture
- Verwendung einer Anwendung (eines System)
  - gleicherweise durch Benutzer und
  - anderen Programmen (u.a. auch automatisierten Tests)
- $\rightarrow$ Entwicklung: unabhängig von eingesetzter Umgebung
  - d.h. anderen Programmen, Datenbanken,…



http://alistair.cockburn.us/Hexagonal+architecture

Ports and Adapters – 2

- andere Ansicht



http://www.dossier-andreas.net/software_architecture/ports_and_adapters.html

- Warum?
  - oft wandert Business Logic in die PL $\rightarrow$ Probleme (Änderungen der UI, Testen, Ablauf im Batchbetrieb bzw. mit Daten eines anderen Prozesses)

Middleware

- Schicht zwischen Schicht 7 und Anwendung
  - erstreckt sich über mehrere Maschinen
    - Begriff auch allgemeiner in SW
- stellt zusätzliche Dienste zur Verfügung
  - Kommunikation: RPC, RMI, Webservice
  - Namensgebung
  - Persistenz und verteilte Transaktionen
- Beispiele: CORBA, ICE, JEE, .NET