Eigenschaften Verteilte Systeme

* Komponenten in eigenen Betriebssytsemprozessen
* Hoch komplexe Systeme
* Offenheit: Lebenszyklus (Wie lange lebt Software?); sobald Software nicht mehr erweiterbar ist, ist der Code tot (30% Entwicklung, 70% Wartung)
  + Abstraktion: Metriken einhalten (Koppelung, Kohäsion)
  + Portabilität: Ausführung auf verschiedenen Umgebungen (z.B.: Betriebsysteme)
  + Interoperabilität: unterschiedliche Services können in unterschiedlichen Frameworks programmiert werden (Node, .NET); Schnittstelle entfernt Restriktionen bei der Wahl des Systems
* Skalierbarkeit: adaptiert sich an die Last der Nutzer (1 Million Nutzer im selben Zeitpunkt); Big Data
  + Scale by splitting by different: Microservice; SOA (unterschiedliche, autonome Komponenten auf unterschiedliche Rechner aufteilen);
  + Scale by splitting by similar: Gesamtheit mittels Gruppierung der homogenen Datensätze (erster Server nur Personen mit A, zweiter Server nur Personen mit B, …)
  + Scale by cloning: Hardwarescaling
  + Bei Hardwareskalierung (vertikal, horizontal): Pivopunkt (wenn shared state), ab welchem die Skalierung nichts mehr bringt
  + Bei Softwareskalierung: kein Pivopunkt, jedoch braucht man Hardwareskalierung
* Transparenz (Tante Erna weiß nur von ihrem Rechner und nicht von den gesamten System im Hintergrund): Abstraktion: komplexes Problem stark abstrahiert, da man nicht mehr die einzelnen Komponenten sieht. „Das System stellt sich für dich einfach dar.“
  + Unterschiedliche Arten: Zugriffstransparenz, Ortstransparenz, Replikationstransparenz; nicht homogen, sondern viele unterschiedliche Felder

Granularität: Microservice: kleine Granularität (ein Service ist noch kleiner); höhere Granularität (ein Service größer); jedes Microservice ist ein SOA, aber nicht jedes SOA ist ein Microservice

Kommunikationsprotokolle

* Regeln, womit die Kommunikation zwischen Partnern geregelt wird
* Unterschiedlicher Aufbau aber gleiche Prinzipien
* HTTP, FTP, UDP, TCP/IP, STOMP, AMQP, MQTT
* Aufteilung der Message: Header (Key-Value), Body (HTTP auch Statuszeile)
* Synchronisation von Nachrichten: One-Way (asynchron) / Fire and Forget, Request-Respone (synchron), Solicit-Response (asynchron) Beispiel: Streaming, Notification (asynchron) Beispiel: AJAX, bidirectional (asynchron)

Message Broker

* Producer -> MB -> Consumer
* High traffic system: problem: wenn ein kommunikationspartner ausfällt, macht es keinen Sinn, dass Producer noch weitere Nachrichten sendet
* MB ist verantwortlich, die Messages zu delivern
* Entkoppelung (Kommunikation exklusiv über Message Broker)
* Asynchrone Kommunikation (Producer wartet ned bis Message bei MB ankommen ist)
* Aufgaben:
  + Persistiert Nachrichten
  + „Router“ für Nachrichten
  + Ausfallsicher
* Kommunikationsformen (je nach MB anders ausimplementiert):
  + Producer-Consumer: der erster C bekommt das Paket
  + Publish-Subscribe: alle Subscriber bekommen das Paket (Paket wird kopiert)
* RabbitMQ:
  + Artefakte: Exchange + Queue
  + Exchange ist der Router des Systems (Wie wird meine Nachricht an nächste Position weitergegeben?); Verschachtlung mehrere Exchanges (ein Exchange vor einem anderen)
  + Queue: immer Exchange davor, aber keine Queue
  + Message hat Messagekey
  + Zwischen Exchange und Queue Bindingkey / Routingkey
  + Vergleich zwischen Messagekey und Bindingkey; je nach Exchangetypen wird der Vergleich unterschiedlich implementiert
  + Exchangetypen (implementiert quasi aus: “bool Equals(object messageKey)”):
    - Direct: ==
    - Topic: # (mehrere Wörter) und \* (exakt ein Wort)
    - Header: Nachricht wie HTTP aufgebaut; Header kann beliebig viele Key-Value haben; x-match: all/any
    - Fanout (Broadcast): Equals -> true