

Modelle und Architekturen verteilter Anwendungen

Dozent: Prof. Dr. Michael Eichberg
Kontakt: michael.eichberg@dhbw-mannheim.de
Version: 2024-02-21



1. MIDDLEWARE

Prof. Dr. Michael Eichberg

Was ist Middleware?

Definition

Middleware ist eine Klasse von Software-Technologien, die dazu dienen,

- I. die Komplexität und
- II. die Heterogenität verteilter Systeme zu verwalten.

Ein einfacher Server mit Sockets

```
/* A simple TCP based server. The port number is passed as an argument */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

void error(char *msg){perror(msg); exit(1);}

int main(int argc, char *argv[]){
    int sockfd, newsockfd, portno, clilen;
    char buffer[256]; int n;
    struct sockaddr_in serv_addr, cli_addr;

    sockfd = socket(AF_INET, SOCK_STREAM, 0); // socket() returns a socket descriptor
    if (sockfd < 0)
        error("ERROR opening socket");

    bzero((char *) &serv_addr, sizeof(serv_addr)); // bzero() sets all values to zero.
    portno = atoi(argv[1]); // atoi() converts str into an integer

    ...
```

1

```
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = htons(portno);

if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
    error("ERROR on binding");
listen(sockfd,5); // tells the socket to listen for connections
clilen = sizeof(cli_addr);
newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
if (newsockfd < 0) error("ERROR on accept");

bzero(buffer,256);
n = read(newsockfd,buffer,255);
if (n < 0) error("ERROR reading from socket");
printf("Here is the message: %s\n",buffer);
n = write(newsockfd,"I got your message",18);

if (n < 0) error("ERROR writing to socket");

return 0;
}
```

2

Ein einfacher Client mit Sockets

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

void error(char *msg){ perror(msg);exit(0);}

int main(int argc, char *argv[]){
    int sockfd, portno, n;
    struct sockaddr_in serv_addr;
    struct hostent *server;
    char buffer[256];

    portno = atoi(argv[2]);

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        error("ERROR opening socket");

    ...
```

1

```
...

server = gethostbyname(argv[1]);
bzero((char *) &serv_addr, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
bcopy((char *)server->h_addr, (char *)&serv_addr.sin_addr.s_addr, server->h_length);
serv_addr.sin_port = htons(portno);

if (connect(sockfd,&serv_addr,sizeof(serv_addr)) < 0) error("ERROR connecting");

printf("Please enter the message: ");
bzero(buffer,256);
fgets(buffer,255,stdin);
n = write(sockfd,buffer,strlen(buffer));
if (n < 0) error("ERROR writing to socket");
bzero(buffer,256);
n = read(sockfd,buffer,255);
printf("%s\n",buffer);

return 0;
}
```

2

Probleme bei der Verwendung von Sockets

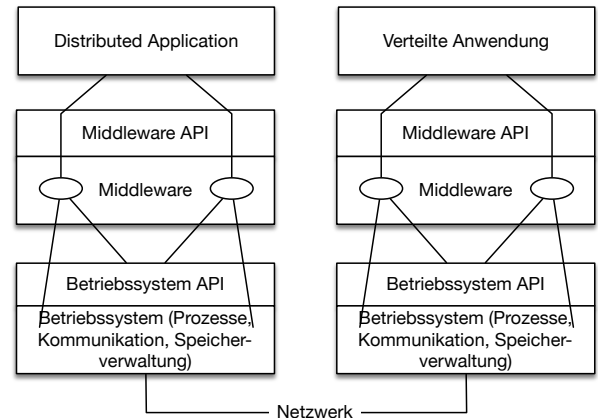
Wir müssen uns um ...

- ! ... die Einrichtung eines Kanals und alle Fehler, die während dieses Prozesses auftreten können
- ! ... die Festlegung eines Protokolls (Wer sendet was, wann, in welcher Reihenfolge und welche Antwort wird erwartet?)
- ! ... Nachrichtenformate (Umwandlung von Daten der Anwendungsebene in Bytes, die über das Netz übertragen werden können).

kümmern!

Middleware als Programmierabstraktion

- Eine Softwareschicht oberhalb des Betriebssystems und unterhalb des Anwendungsprogramms, die eine gemeinsame Programmierabstraktion in einem verteilten System bietet.
- Ein Baustein auf höherer Ebene als die vom Betriebssystem bereitgestellten APIs (z.B. Sockets)



Middleware als Programmierabstraktion

Die von Middleware angebotenen Programmierabstraktionen verbergen einen Teil der Heterogenität und bewältigen einen Teil der Komplexität, mit der Programmierer einer verteilten Anwendung umgehen müssen:

- ✓ Middleware maskiert immer die Heterogenität der zugrundeliegenden Netzwerke und Hardware.
- ✓ Middleware maskiert meistens die Heterogenität von Betriebssystemen und/oder Programmiersprachen.
- ✓ Manche Middleware maskiert sogar die Heterogenität zwischen den Implementierungen des gleichen Middleware-Standards durch verschiedene Hersteller.

Alte Middlewarestandards wie zum Beispiel CORBA waren sehr komplex und die Implementierungen verschiedener Hersteller meist nicht vollständig kompatibel.

Transparenzziele von Middleware aus Sicht der Programmierung

Middleware bietet (beim Programmieren) Transparenz in Bezug auf eine oder mehrere der folgenden Dimensionen:

- Standort
- Nebenläufigkeit
- Replikation
- Ausfälle (bedingt)

Middleware ist die Software, die ein verteiltes System (DS) programmierbar macht.

Middleware als Infrastruktur

- Hinter Programmierabstraktionen steht eine komplexe Infrastruktur, die diese Abstraktionen implementiert
(Middleware-Plattformen können sehr komplexe Softwaresysteme sein).
- Da die Programmierabstraktionen immer höhere Ebenen erreichen, muss die zugrunde liegende Infrastruktur, die die Abstraktionen implementiert, entsprechend wachsen.
- Zusätzliche Funktionalität wird fast immer durch zusätzliche Softwareschichten implementiert.
- Die zusätzlichen Softwareschichten erhöhen den Umfang und die Komplexität der für die Nutzung der neuen Abstraktionen erforderlichen Infrastruktur.

Seit Jahrzehnten kann beobachtet werden, dass Middleware immer komplexer wird bzw. wurde bis zu dem Punkt an dem die Komplexität kaum mehr beherrschbar war. Zu diesen Zeitpunkten wurden dann häufig neue Ansätze entwickelt, die die Komplexität reduzierten bis diese wiederum Eingang in komplexere Middleware-Produkten fand.

Ansätze, wie z.B. REST, haben sich als recht erfolgreich erwiesen stellen aber Entwickler vor neue Herausforderungen.

Middleware und nicht-funktionale Anforderungen

Die Infrastruktur kümmert sich um nicht-funktionale Eigenschaften, die normalerweise von Datenmodellen, Programmiermodellen und Programmiersprachen ignoriert werden:

- Leistung
- Verfügbarkeit
- Ressourcenmanagement
- Zuverlässigkeit
- usw.

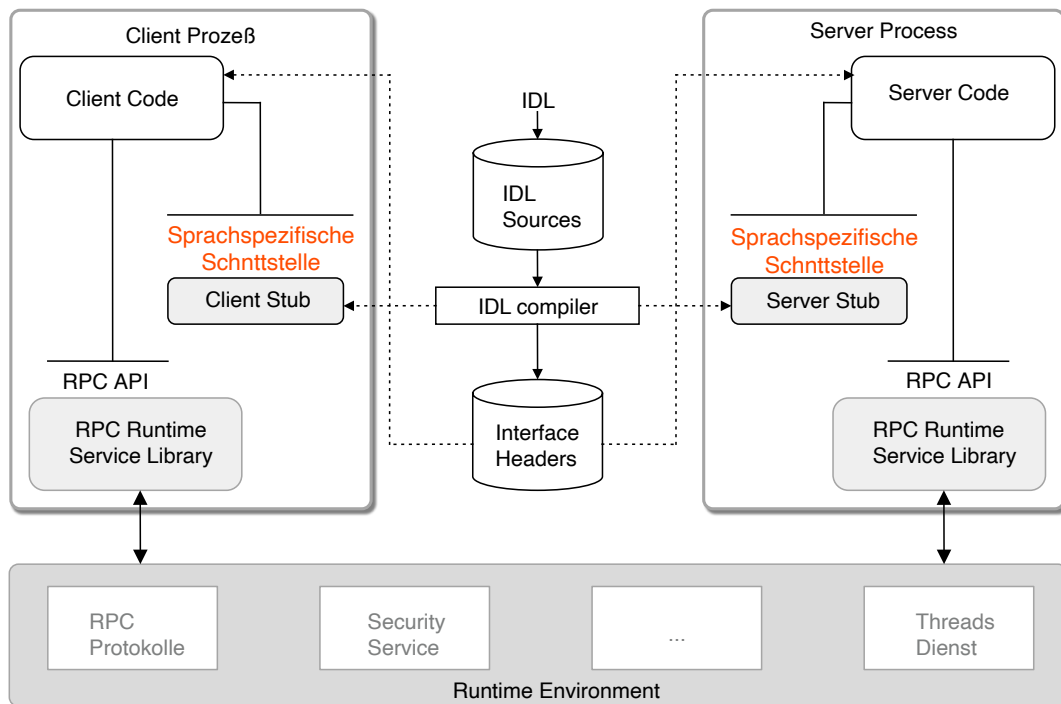
Middleware als Infrastruktur

Middleware unterstützt zusätzliche Funktionen die die Entwicklung, Wartung und Überwachung einfacher und kostengünstiger machen (Auszug):

- Protokollierung (📄 *Logging*)
- Wiederherstellung (📄 *Recovery*)
- Sprachprimitive für transaktionale Abgrenzung

bzw. fortgeschrittene Transaktionsmodelle (z.B. transaktionale RPC),
transaktionales Dateisystem

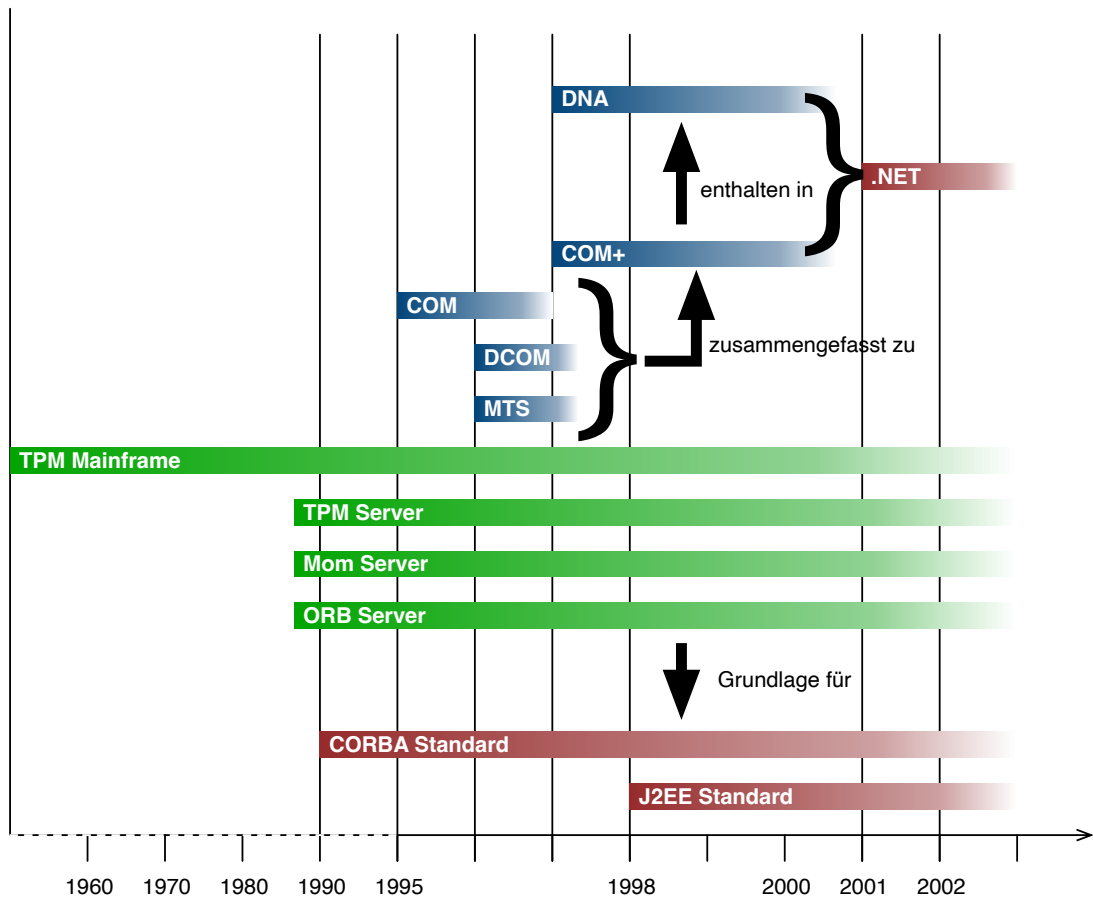
Konzeptionelle Darstellung (historischer) Middleware



Darstellung nach: Alonso; Web services: Concepts, Architectures and Applications; Springer, 2004

Insbesondere die explizite Erzeugung von Stubs und Skeletons durch einen IDL Compiler erfolgt so in der heutigen Zeit nicht mehr. Die Erzeugung von Stubs und Skeletons - wenn überhaupt erforderlich - erfolgt heute automatisch durch die Middleware.

Historische Entwicklung von Middleware



Entwicklung von Middleware

- Beabsichtigt, Details der Hardware, der Netze und der Verteilung auf niedriger Ebene zu verbergen.
- Anhaltender Trend zu immer leistungsfähigeren Primitiven (*Events*), die zusätzliche Eigenschaften haben oder eine flexiblere Nutzung des Konzepts ermöglichen.
- Die Entwicklung und das Erscheinungsbild für den Programmierer wird von den Trends in den Programmiersprachen diktiert:
 - RPC und C
 - CORBA und C++
 - RMI (Corba) und Java
 - "Klassische" Webservices und XML
 - RESTful Webservices und JSON

Eine Middleware stellt eine umfassende Plattform für die Entwicklung und den Betrieb komplexer verteilter Systeme zur Verfügung.

Remote Procedure Calls (RPCs)

Remote Procedure Call (RPC)

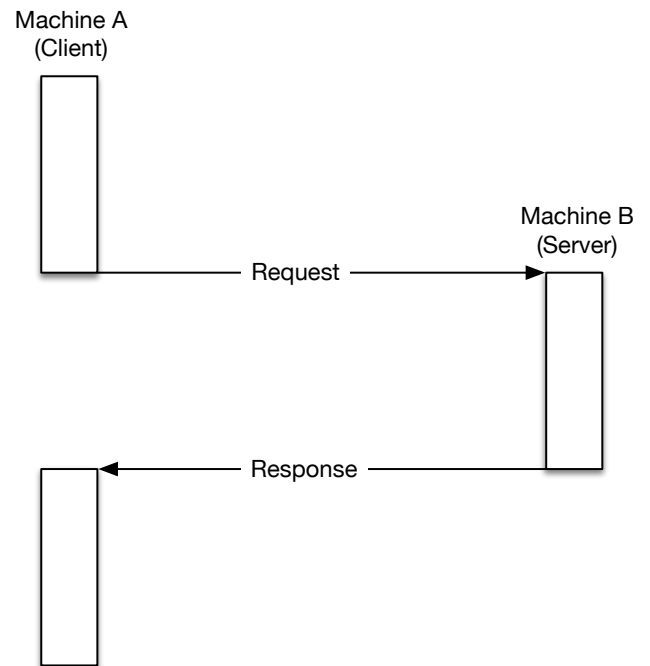
Schwerpunkt: verstecken der Netzkommunikation.

Ein Prozess kann eine Prozedur aufrufen kann, deren Implementierung sich auf einem entfernten Rechner befindet:

- Der Programmierer von verteilten Systemen muss sich nicht mehr um alle Details der Netzwerkprogrammierung kümmern (d.h. keine "expliziten" Sockets mehr)
- Überbrückung der konzeptionellen Lücke zwischen dem Aufruf lokaler Funktionalität über Prozeduren und dem Aufruf entfernter Funktionalität über Sockets.

RPCs konzeptionell (synchrone Kommunikation)

- Ein Server ist ein Programm, das bestimmte Dienste implementiert.
- Clients möchten diese Dienste in Anspruch nehmen:
 - Die Kommunikation erfolgt durch das Senden von Nachrichten (kein gemeinsamer Speicher, keine gemeinsamen Festplatten usw.)
 - Einige minimale Garantien müssen gegeben werden (Behandlung von Fehlern, Aufrufsemantik, usw.)



RPCs - zentrale Fragestellungen und Herausforderungen

Sollen entfernte Aufrufe transparent oder nicht transparent für den Entwickler sein?

Ein Fernaufruf ist etwas völlig anderes als ein lokaler Aufruf; sollte sich der Programmierer dessen bewusst sein?

1

Wie können Daten zwischen Maschinen ausgetauscht werden, die möglicherweise unterschiedliche Darstellungen für verschiedene Datentypen verwenden?

2

Komplexe Datentypen müssen linearisiert werden:

- **Marshalling** - der Prozess des Aufbereitens der Daten in eine für die Übermittlung in einer Nachricht geeignete Form
- **Unmarshalling** - der Prozess der Wiederherstellung der Daten bei ihrer Ankunft am Zielort, um eine originalgetreue Repräsentation zu erhalten.

3

Wie findet und bindet man den Dienst, den man tatsächlich will, in einer potenziell großen Sammlung von Diensten und Servern?

Das Ziel ist, dass der Kunde nicht unbedingt wissen muss, wo sich der Server befindet oder sogar welcher Server den Dienst anbietet (Standorttransparenz).

4

Wie geht man mehr oder weniger elegant mit Fehlern um:

- Server ist ausgefallen
- Kommunikation ist gestört
- Server beschäftigt
- doppelte Anfragen ...

5

Je nach System ist die Reihenfolge der Bytes unterschiedlich:

- Intel-CPU's sind Little-Endian.
- PowerPC ist Big-Endian.
- ARM kann beides und ist meistens Little-Endian.

High-level View auf RPC

Für Programmierer sieht ein „entfernter“ Prozeduraufruf fast identisch aus wie ein „lokaler“ Prozeduraufruf und funktioniert auch so - auf diese Weise wird Transparenz erreicht.

Um Transparenz zu erreichen, führte RPC viele Konzepte von Middleware-Systemen ein:

- *Interface Description Language* (IDL)
- Verzeichnis- und Benennungsdienste
- Dynamische Bindung
- Marshalling und Unmarshalling
- *Opaque References*, um bei verschiedenen Aufrufen auf dieselbe Datenstruktur oder Entität auf dem Server zu verweisen.

(Der Server ist für die Bereitstellung dieser undurchsichtigen Verweise verantwortlich.)

RPC - Call Semantics

Nehmen wir an, ein Client stellt eine RPC-Anfrage an einen Dienst eines bestimmten Servers. Nachdem die Zeitüberschreitung abgelaufen ist, beschließt der Client die Anfrage erneut zu senden. Das finale Verhalten hängt von der Semantik des Aufrufs (🇺🇸 *Call Semantics*) ab:

Maybe (vielleicht; keine Garantie)

Die Zielmethode kann ausgeführt worden sein und die Antwortnachricht(en) ging(en) verloren oder die Methode wurde gar nicht erst ausgeführt da die Anfrage verloren ging.

XMLHttpRequests in Webbrowsern verwenden diese Semantik.

1

At least once (mindestens einmal)

Die Prozedur wird ausgeführt werden solange der Server nicht endgültig versagt.

Es ist jedoch möglich, dass sie mehr als einmal ausgeführt wird wenn der Client die Anfrage nach einer Zeitüberschreitung erneut gesendet hatte.

2

At most once (höchstens einmal)

Die Prozedur wird entweder einmal oder gar nicht ausgeführt. Ein erneutes Senden der Anfrage führt nicht dazu, dass die Prozedur mehrmals ausgeführt wird.

3

Exactly once (genau einmal)

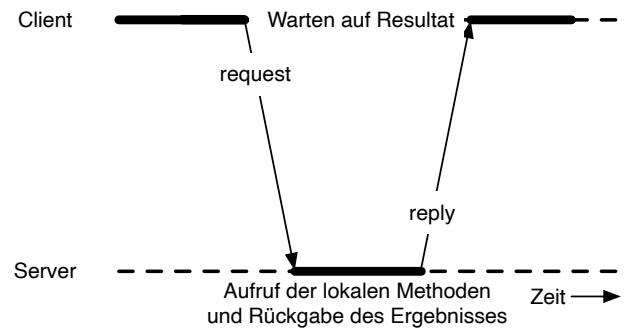
Das System garantiert die gleiche Semantik wie bei lokalen Aufrufen unter der Annahme, dass ein abgestürzter Server irgendwann wieder startet.

Verwaiste Aufrufe, d.h. Aufrufe auf abgestürzten Server-Rechnern, werden nachgehalten, damit sie später von einem neuen Server übernommen werden können.

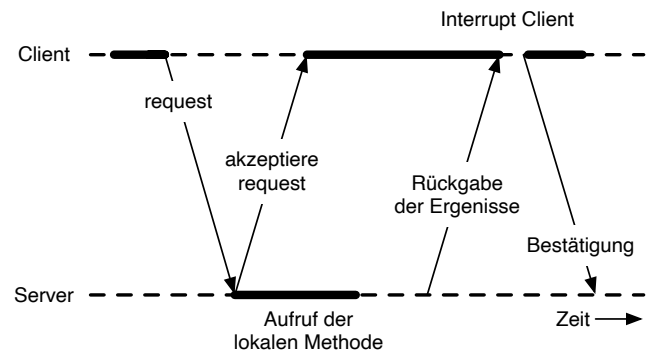
4

Asynchrones RPC

Die Verbindung zwischen Client und Server in einem traditionellen RPC. Der Client wird blockiert und wartet.



Die Verbindung zwischen Client und Server bei einem asynchronen RPC. Der Client wird nicht blockiert.



Ein normaler Aufruf mittels `XMLHttpRequest` (JavaScript) ist auch immer asynchron.

RPC - Bewertung

- ✓ RPC bot einen Mechanismus, um verteilte Anwendungen auf einfache und effiziente Weise zu implementieren.
- ✓ RPC ermöglichte den modularen und hierarchischen Aufbau großer verteilter Systeme:
 - Client und Server sind getrennte Einheiten
 - Der Server kapselt und verbirgt die Details der Backend-Systeme (wie z.B. Datenbanken)
- ! RPC ist kein Standard, sondern wurde auf viele verschiedene Arten umgesetzt wurde.
- ! RPC ermöglicht Entwicklern den Aufbau verteilter Systeme, löst aber nur ausgewählte Aspekte.

Wenn man moderne Ansätze wie RESTful WebServices mit RPC vergleicht, dann fällt auf, dass RPC eine deutlich bessere Transparenz bietet.

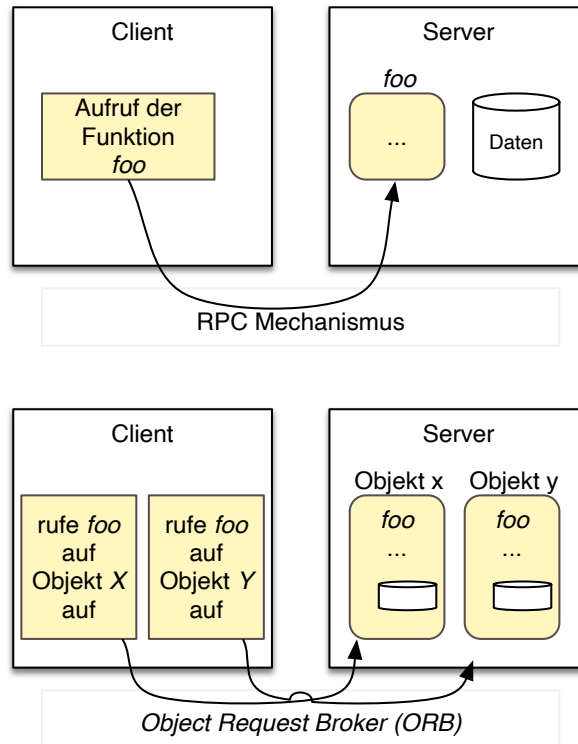
Java Remote Method Invocation (RMI)

Java RMI (Remote Method Invocation)

Ermöglicht es einem Objekt, das in einer Java Virtual Machine (VM) läuft, Methoden eines Objekts aufzurufen, das in einer anderen Java VM läuft.

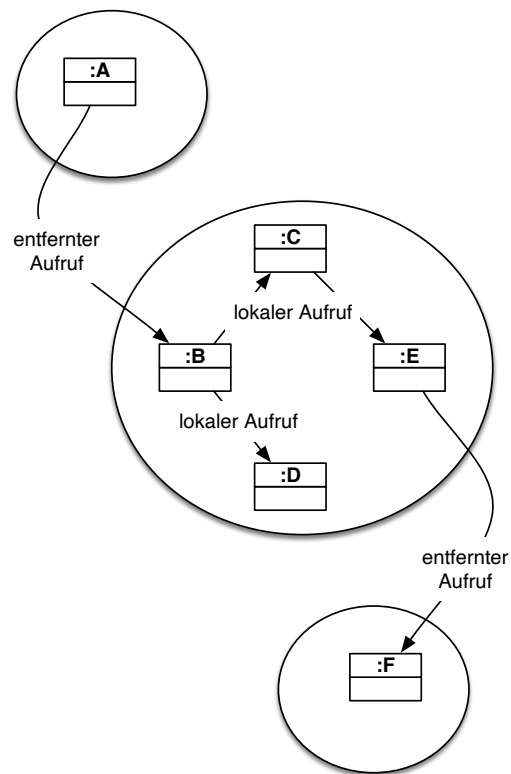
- Entfernte Objekte können ähnlich wie lokale Objekte behandelt werden.
- Übernimmt das Marshalling, den Transport und die Garbage Collection der entfernten Objekte.
- Teil von Java seit JDK 1.1

Java RMI vs. RPC



Java RMI ist eine spezielle Form von RPC, die in Java implementiert wurde. Der Unterschied ergibt sich im Prinzip aus dem Unterschied zwischen einem Prozeduraufruf und einem Methodenaufruf auf ein Objekt

Java RMI implementiert ein *Distributed Object Model*



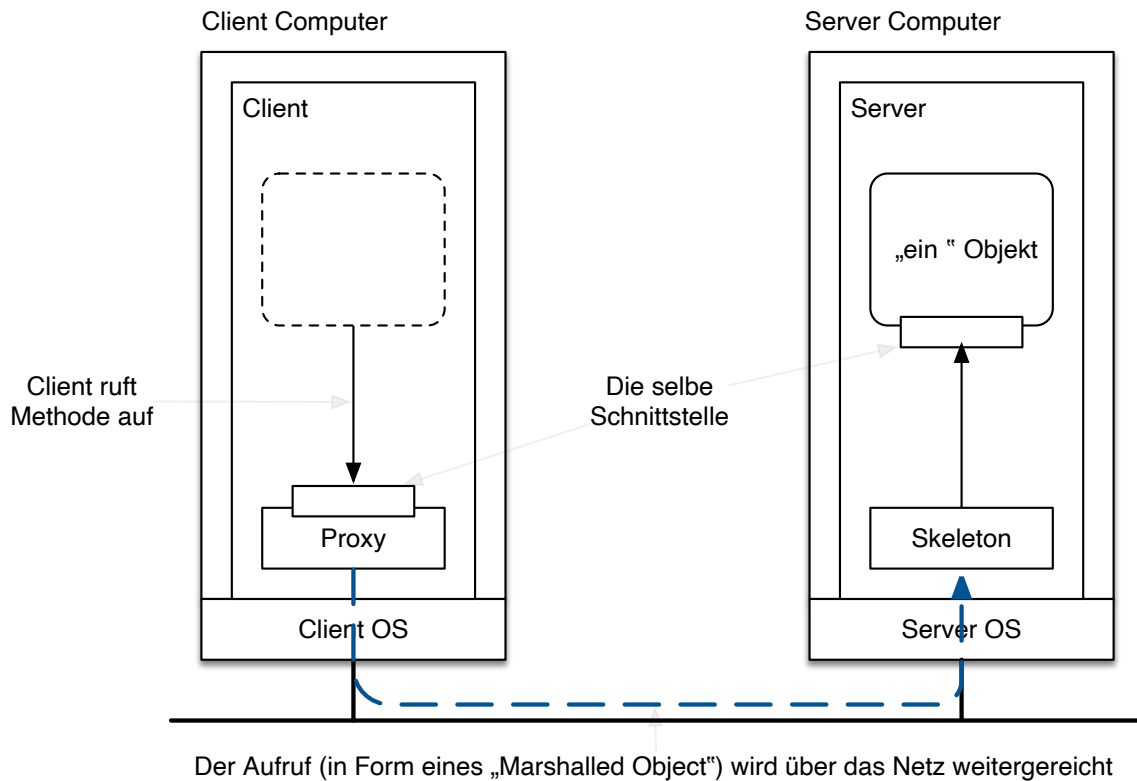
28

- Jeder Prozess enthält sowohl Objekte die entfernte Aufrufe empfangen können als auch solche, die nur lokale Aufrufe empfangen können.

(Objekte die entfernte Aufrufe empfangen können, werden *Remote Objects* genannt).

- Objekte müssen die Remote-Objektreferenz eines Objekts in einem anderen Prozess kennen, um dessen Methoden aufrufen zu können (Remote Method Invocation; Remote Object References)

Anatomie eine Java RMI Aufrufs



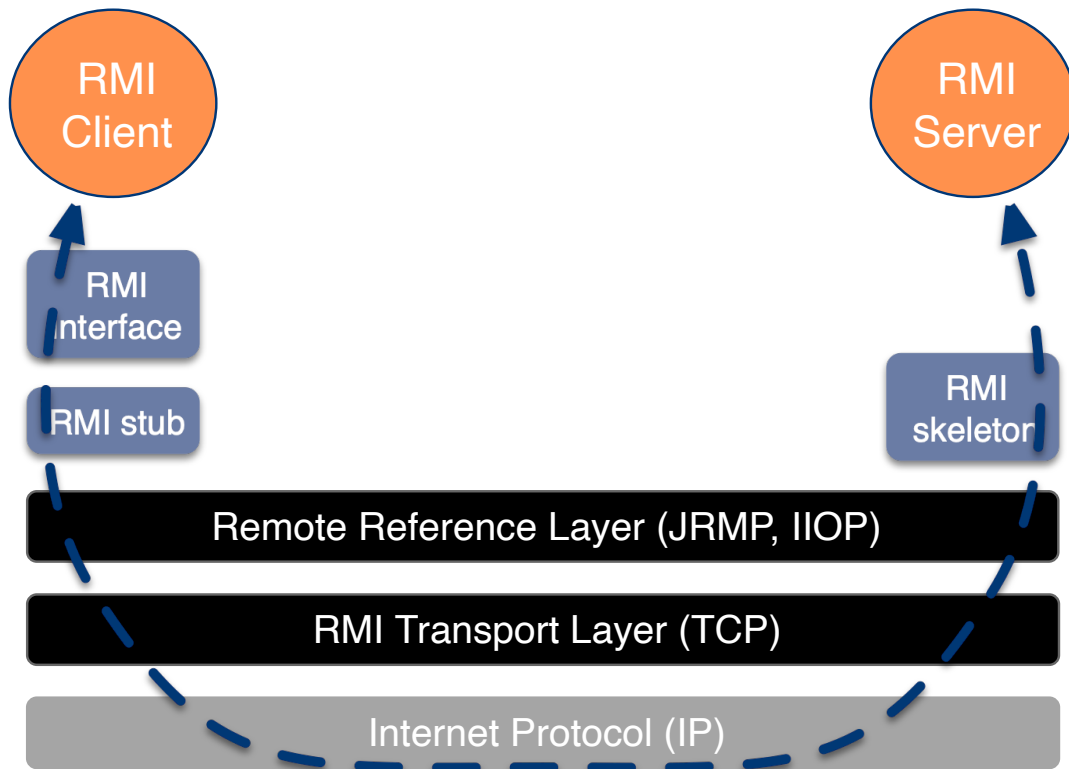
29

Der Proxy versteckt für den Client, dass es sich um einen entfernten Aufruf handelt. Er implementiert die Remote-Schnittstelle und kümmert sich um das Marshalling und Unmarshalling der Parameter und des Ergebnisses.

Der Skeleton ist für die Entgegennahme der Nachrichten verantwortlich und leitet die Nachricht an das eigentliche Objekt weiter. Er sorgt für die Transparenz auf Serverseite.

Referenzen auf *Remote Objects* sind systemweit eindeutig und können frei zwischen Prozessen weitergegeben werden (z.B. als Parameter). Die Implementierung der entfernten Objektreferenzen wird von der Middleware verborgen (*Opaque-Referenzen*).

RMI Protocol Stack



30

- *Remote Reference Layer*: RMI-spezifische Kommunikation über TCP/IP, Verbindungsinitialisierung, Serverstandort, Verarbeitung serialisierter Daten
- *RMI Transport Layer (TCP)*: Verbindungsverwaltung, Bereitstellung einer zuverlässigen Datenübertragung zwischen Endpunkten
- Internetprotokoll in IP-Paketen enthaltene Datenübertragung (unterste Ebene)

Einfacher RMI Dienst und Aufruf

Schnittstelle des Zeitserver

```
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.Date;

public interface Time extends Remote {
    public Date getTime() throws RemoteException;
}
```

1

Implementierung der Schnittstelle durch den Zeitserver

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.Date;

public class TimeServer extends UnicastRemoteObject implements Time {
    public TimeServer() throws RemoteException {
        super();
    }

    public Date getTime() {
        return new Date();
    }
}
```

2

Registrierung des Zeitserver

```
import java.rmi.Naming;

public class TimeRegistrar {

    /** @param args args[0] has to specify the hostname. */
    public static void main(String[] args) throws Exception {
        String host = args[0];
        TimeServer timeServer = new TimeServer();
        Naming.rebind("rmi://" + host + "/ServerTime", timeServer);
    }
}
```

3

Client des Zeitserver

```
import java.rmi.Naming;
import java.util.Date;
```



```
public class TimeClient {  
    public static void main(String[] args) throws Exception {  
        String host = args[0];  
        Time timeServer = (Time) Naming.lookup("rmi://" + host + "/ServerTime");  
        System.out.println("Time on " + host + " is " + timeServer.getTime());  
    }  
}
```

4

- RMI verwendet einen referenzzählenden Garbage-Collection-Algorithmus. Netzwerkprobleme können dann zu einer verfrühten GC führen was wiederum bei Aufrufen zu Ausnahmen führen kann.
- Die Aufrufsemantik (*Call Semantics*) von RMI ist *at most once*.
- (Un)Marshalling ist in Java RMI automatisch und verwendet Java Object Serialization.

Der Overhead kann leicht ~25%-50% der Zeit für einen entfernten Aufruf ausmachen.

Übung:

System Message: WARNING/2 (ds-architekturen/folien.rst, line 769)

Explicit markup ends without a blank line; unexpected unindent.

2. TODO

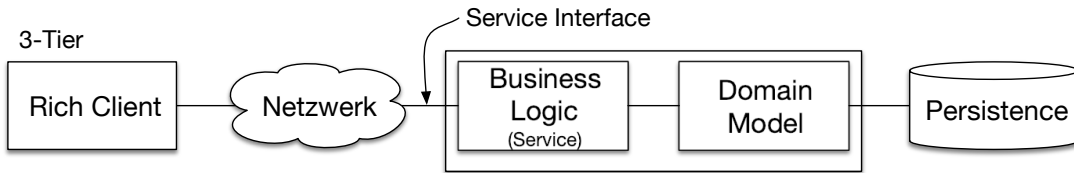
Prof. Dr. Michael Eichberg

Klassische Architekturen

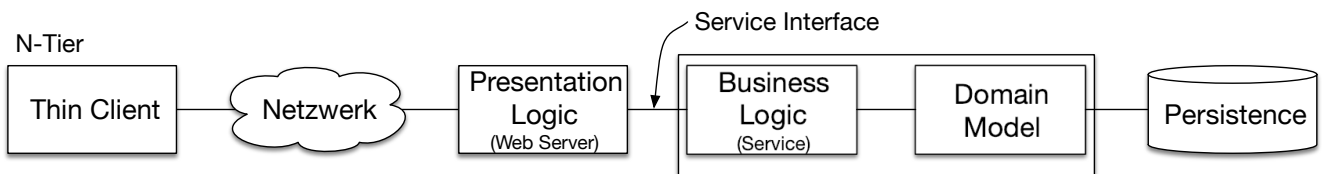
2-Tier



3-Tier



N-Tier



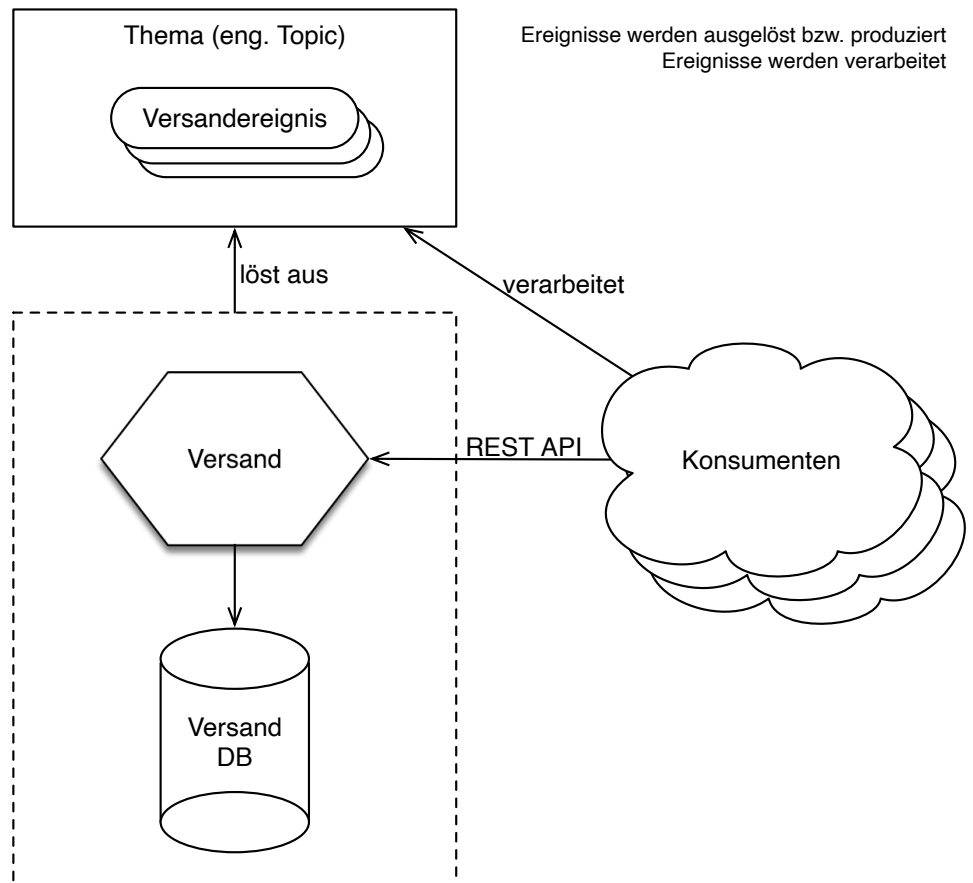
3. MICROSERVICES [NEWMAN2021]

Prof. Dr. Michael Eichberg

Microservices

Ein einfacher
Microservice, der eine
REST Schnittstelle
anbietet und Ereignisse
auslöst.

Wo liegen hier die
Herausforderungen?



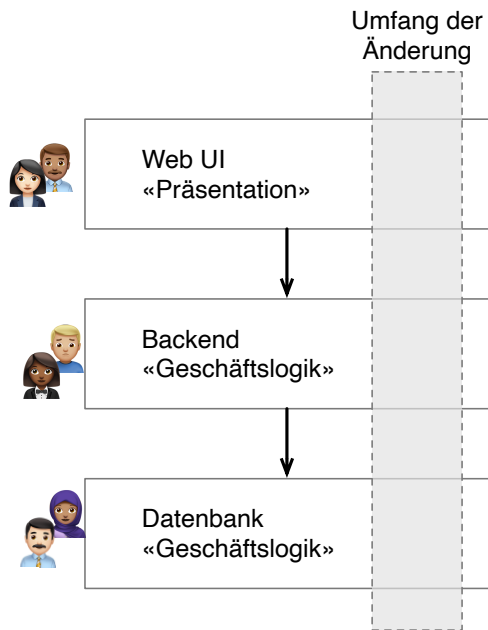
Ein große Herausforderung ist das Design der Schnittstellen. Um wirkliche Unabhängigkeit zu erreichen, müssen die Schnittstellen sehr gut definiert sein. Sind die Schnittstellen nicht klar definiert oder unzureichend, dann kann das zu viel Arbeit und Koordination zwischen den Teams führen, die eigentlich unerwünscht ist!

Schlüsselkonzepte von Microservices

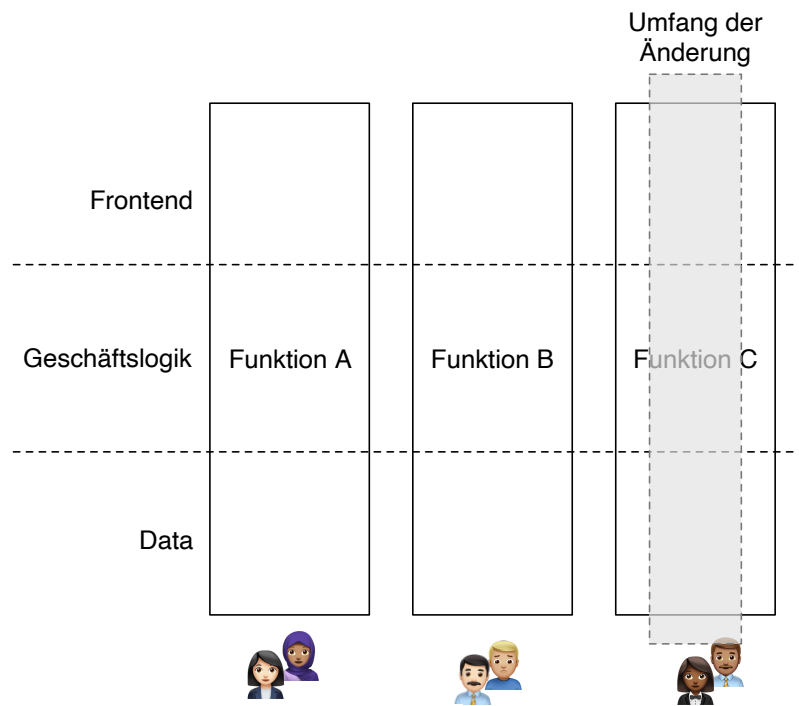
- können unabhängig bereitgestellt werden (🚩 *independently deployable*)
und werden unabhängig entwickelt
- modellieren eine Geschäftsdomäne
Häufig entlang einer Kontextgrenze (eng. Bounded Context) oder eines Aggregats aus DDD
- verwalten Ihren eigenen Zustand
d.h. keine geteilten Datenbanken
- sind klein
Klein genug, um durch (max.) ein Team entwickelt werden zu können
- flexibel bzgl. Skalierbarkeit, Robustheit, eingesetzter Technik
- erlauben das Ausrichten der Architektur an der Organisation (vgl. Conway's Law)

Microservices und Conway's Law

Traditionelle Schichtenarchitektur

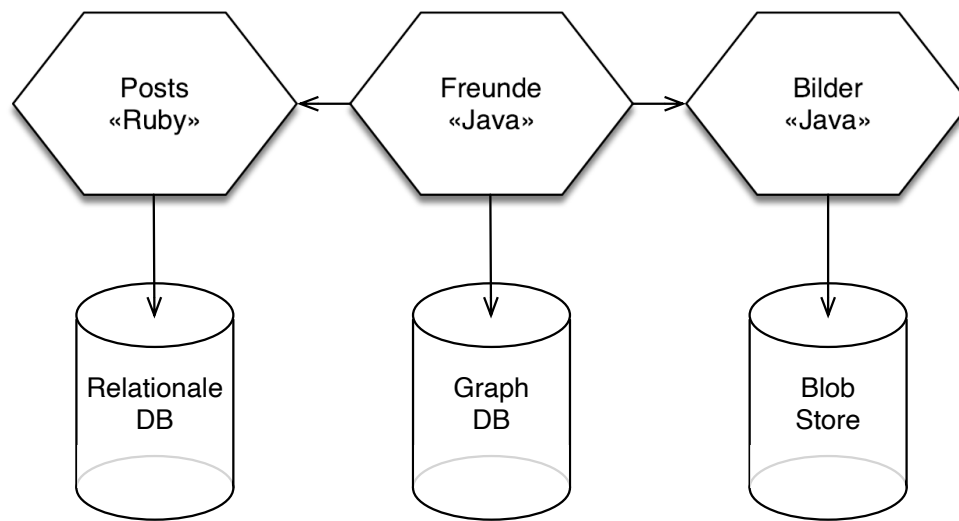


Microservices Architektur






















Microservices und Technologieeinsatz

Microservices sind flexibel bzgl. des Technologieeinsatzes und ermöglichen den Einsatz „der geeignetsten“ Technologie.



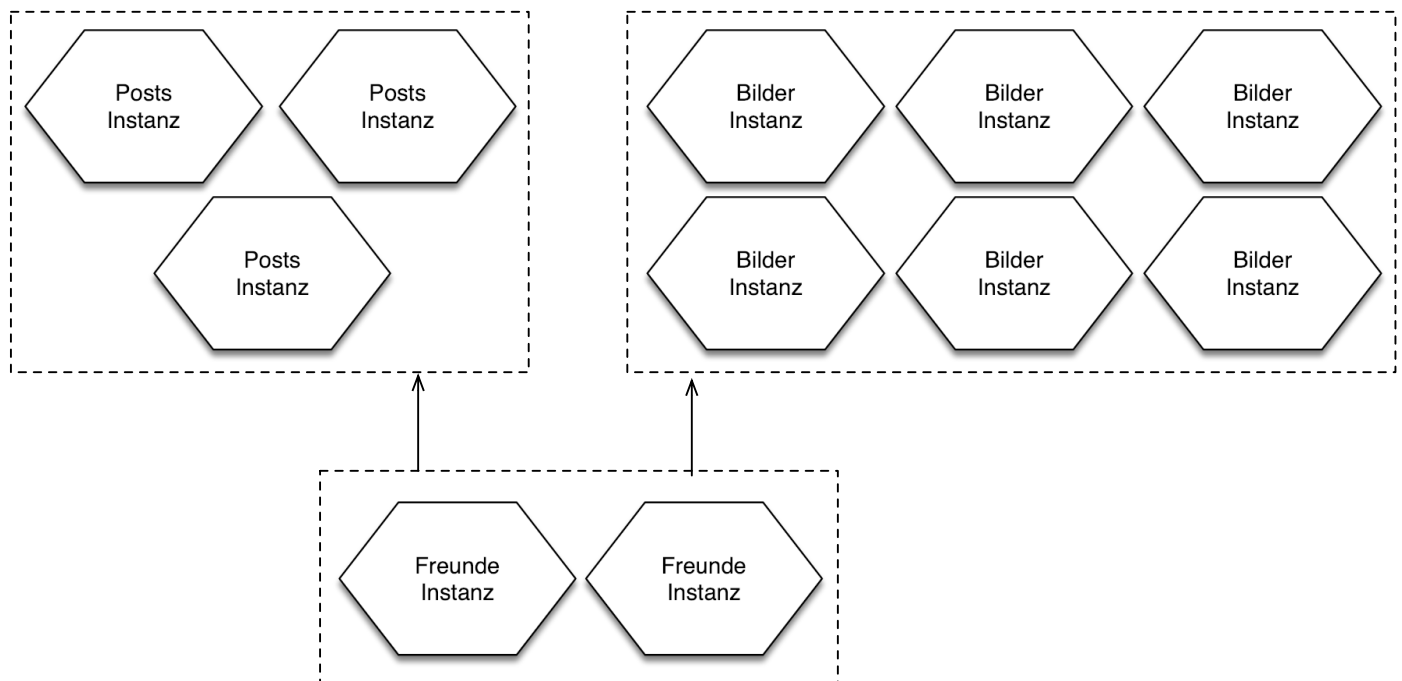
Position Apr 2012	Position Apr 2011	Delta in Position	Programming Language	Ratings Apr 2012	Delta Apr 2011	Status
1	2	↑	C	17.555%	+1.39%	A
2	1	↓	Java	17.026%	-2.02%	A
3	3	=	C++	8.896%	-0.33%	A
4	8	↑↑↑↑	Objective-C	8.236%	+3.85%	A
5	4	↓	C#	7.348%	+0.16%	A
6	5	↓	PHP	5.288%	-1.30%	A
7	7	=	(Visual) Basic	4.962%	+0.28%	A
8	6	↓↓	Python	3.665%	-1.27%	A
9	10	↑	JavaScript	2.879%	+1.37%	A
10	9	↓	Perl	2.387%	+0.40%	A
11	11	=	Ruby	1.510%	+0.03%	A
12	24	↑↑↑↑↑↑↑↑	PL/SQL	1.373%	+0.92%	A
13	13	=	Delphi/Object Pascal	1.370%	+0.34%	A
14	35	↑↑↑↑↑↑↑↑	Visual Basic .NET	0.978%	+0.64%	A
15	15	=	Lisp	0.951%	+0.02%	A
16	17	↑	Pascal	0.812%	+0.10%	A
17	16	↓	Ada	0.783%	+0.01%	A-
18	18	=	Transact-SQL	0.760%	+0.18%	A
19	22	↑↑↑	Logo	0.652%	+0.12%	B
20	52	↑↑↑↑↑↑↑↑	NXT-G	0.578%	+0.35%	B

Quelle: TIOBE Programming Community Index - April 2012

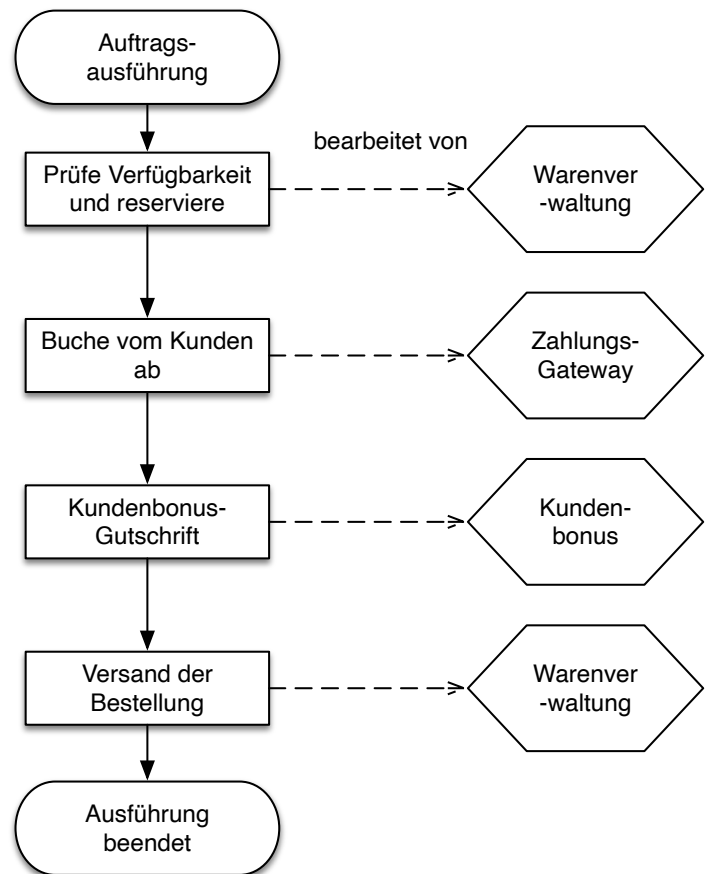
Feb 2024	Feb 2023	Change	Programming Language	Ratings	Change
1	1		 Python	15.16%	-0.32%
2	2		 C	10.97%	-4.41%
3	3		 C++	10.53%	-3.40%
4	4		 Java	8.88%	-4.33%
5	5		 C#	7.53%	+1.15%
6	7	↗	 JavaScript	3.17%	+0.64%
7	8	↗	 SQL	1.82%	-0.30%
8	11	↗	 Go	1.73%	+0.61%
9	6	↘	 Visual Basic	1.52%	-2.62%
10	10		 PHP	1.51%	+0.21%
11	24	↗	 Fortran	1.40%	+0.82%
12	14	↗	 Delphi/Object Pascal	1.40%	+0.45%
13	13		 MATLAB	1.26%	+0.27%
14	9	↘	 Assembly language	1.19%	-0.19%
15	18	↗	 Scratch	1.18%	+0.42%
16	15	↘	 Swift	1.16%	+0.23%
17	33	↗	 Kotlin	1.07%	+0.76%
18	20	↗	 Rust	1.05%	+0.35%
19	30	↗	 COBOL	1.01%	+0.60%

Microservices und Skalierbarkeit

Sauber entworfene Microservices können sehr gut skaliert werden.

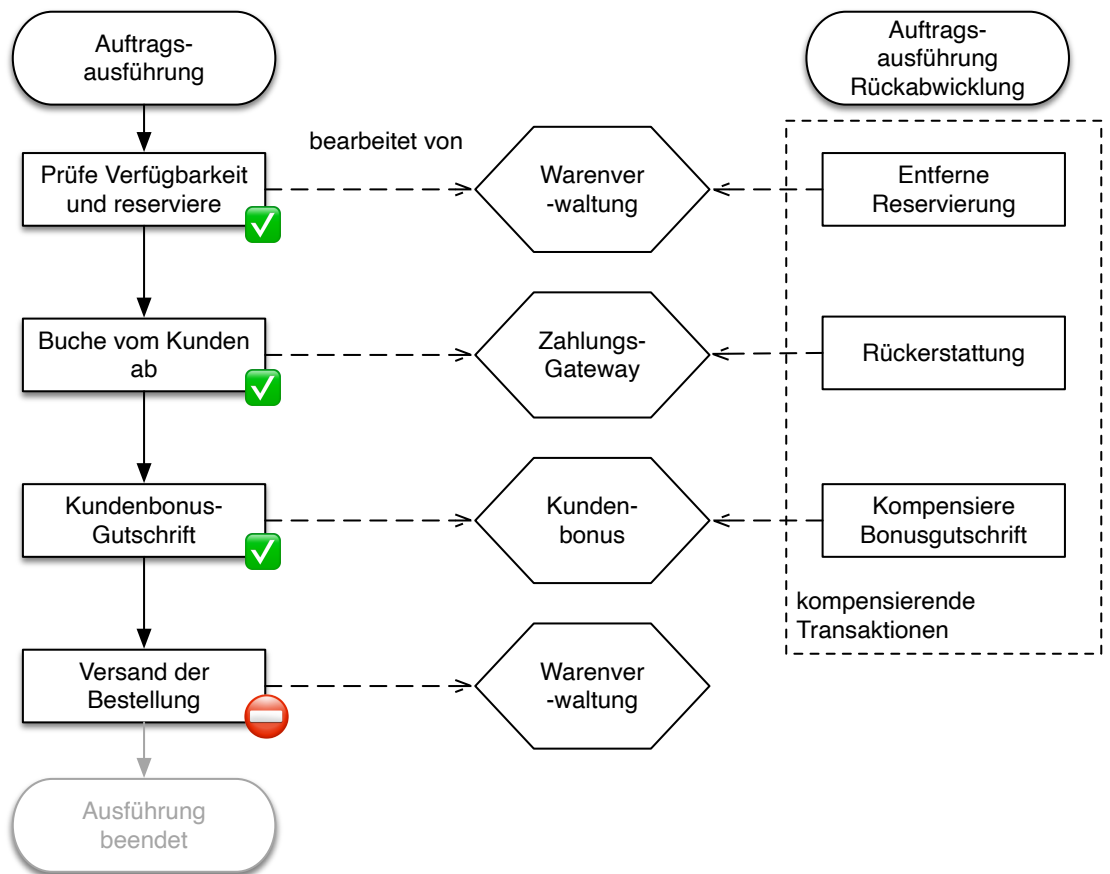


Implementierung einer langlebigen Transaktion?



Die Implementierung von Transaktionen ist eine der größten Herausforderungen bei der Entwicklung von Microservices.

Aufteilung einer langlebigen Transaktion mit Hilfe von Sagas



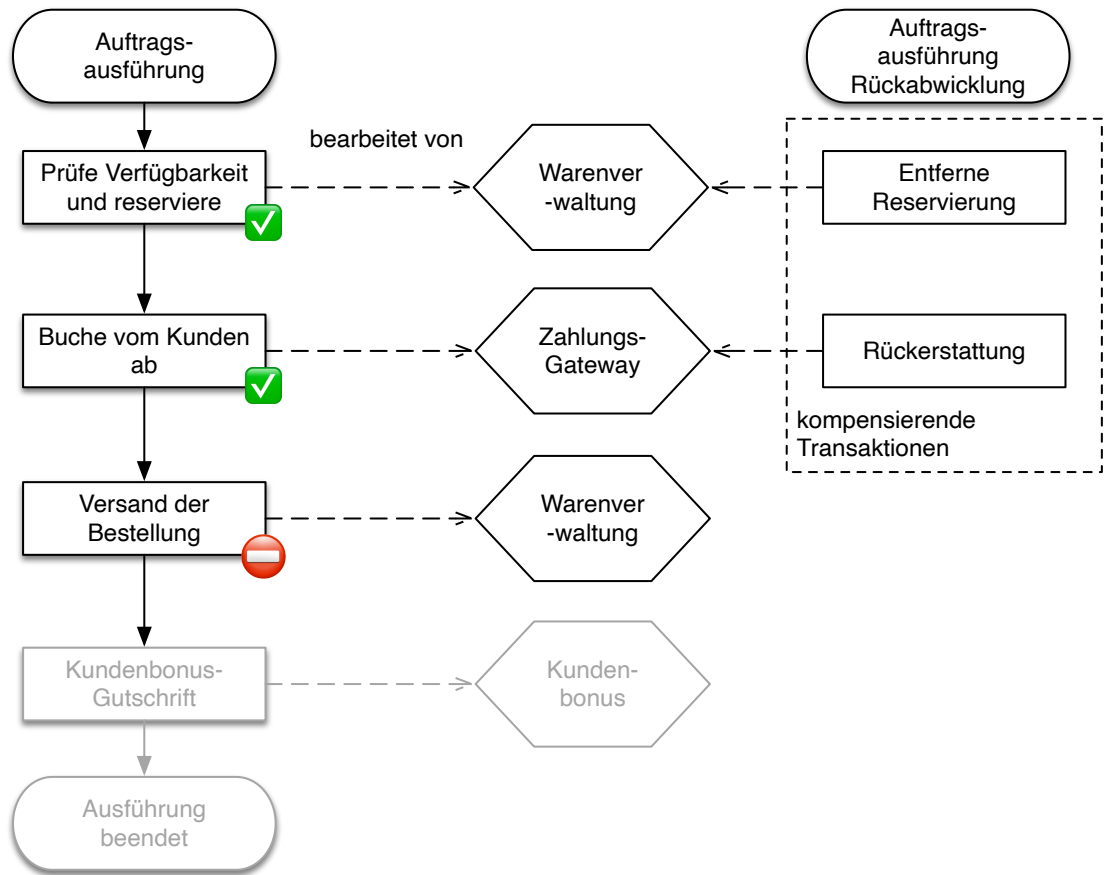
43

Eine *Saga* ist eine Sequenz von Aktionen, die ausgeführt werden, um eine langlebige Transaktion zu implementieren.

Sagas können keine Atomizität garantieren!. Jedes System für sich kann jedoch ggf. Atomizität garantieren (z.B. durch die Verwendung traditioneller Datenbanktransaktionen).

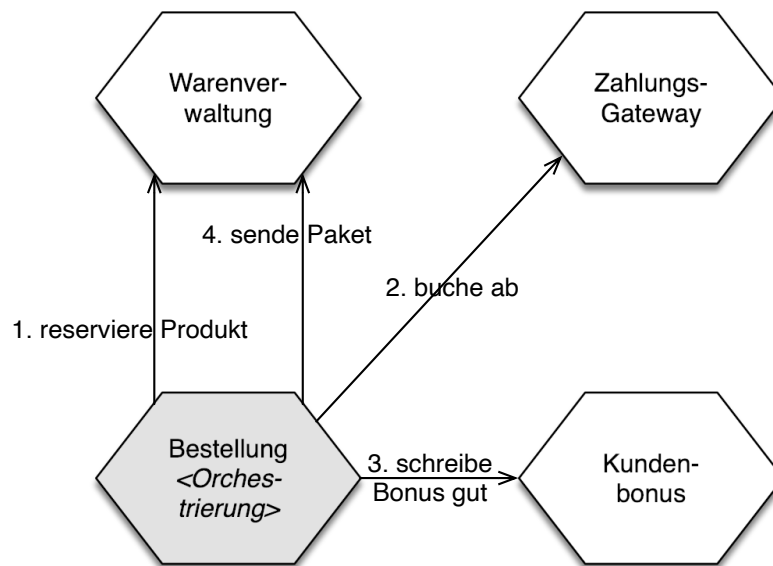
Sollte ein Abbruch der Transaktion notwendig sein, dann kann kein traditioneller *Rollback* erfolgen. Die Saga muss dann entsprechende kompensierende Transaktionen durchführen, die alle bisher erfolgreich durchgeführten Aktionen rückgängig machen.

Optimierung der Abarbeitungsreihenfolge zwecks Minimierung von mgl. *Rollbacks*



Die Abarbeitungsreihenfolge der Aktionen kann so optimiert werden, dass die Wahrscheinlichkeit von *Rollbacks* minimiert wird. In diesem Falle ist die Wahrscheinlichkeit, dass es zu einem *Rollback* während des Schritts „Versand der Bestellung“ kommt, wesentlich höher als beim Schritt „Kundenbonus gutschreiben“.

Langlebige Transaktionen mit orchestrierten Sagas



45

Die orchestrierte Saga ist eine Möglichkeit, um langlebige Transaktionen zu implementieren.

✓ Mental einfach

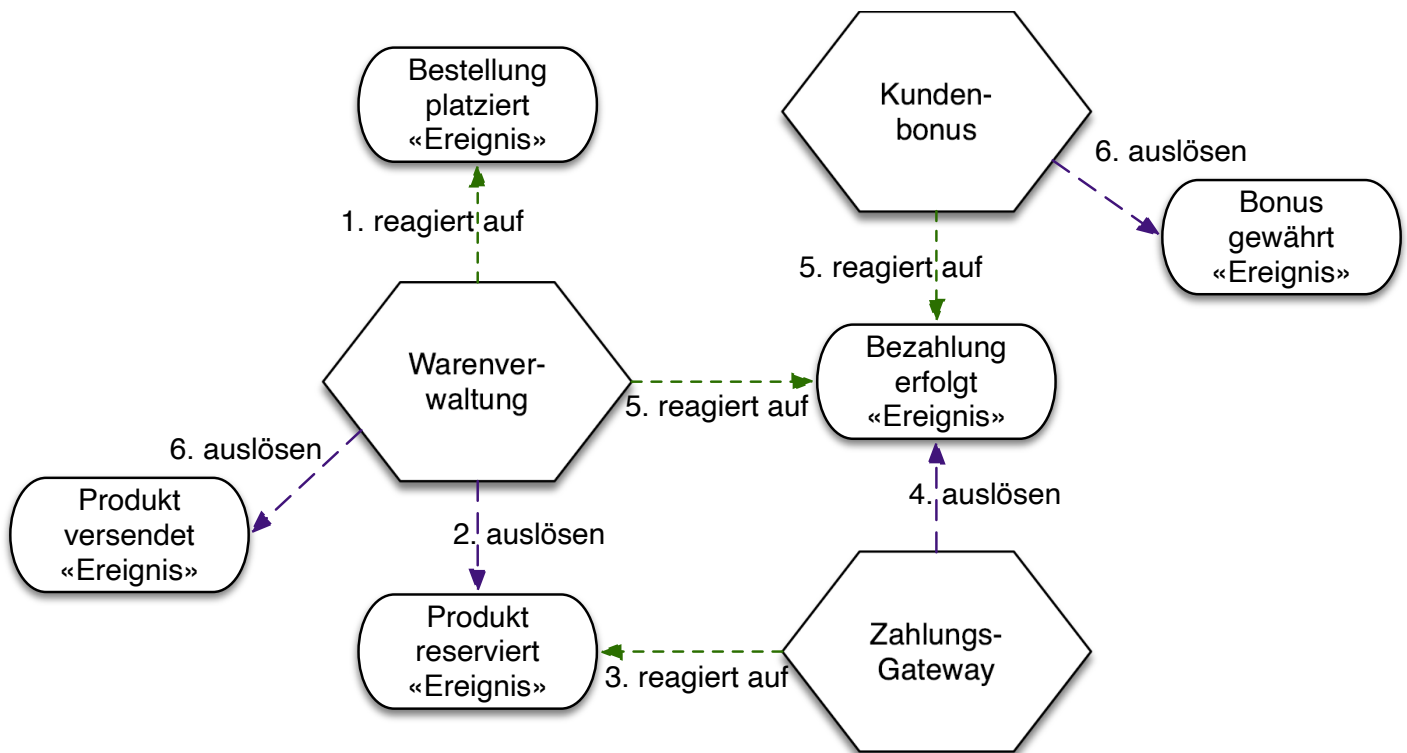
! Hoher Grad an *Domain Coupling*

Da es sich im Wesentlichen um fachlich getriebene Kopplung handelt, ist diese Kopplung häufig akzeptabel. Die Kopplung erzeugt keine technischen Schulden (📦 *technical debt*).

! Hoher Grad an *Request-Response* Interaktionen

! Gefahr, dass Funktionalität, die besser in den einzelnen Services (oder ggf. neuen Services) unterzubringen wäre, in den Bestellung Service wandert.

Langlebige Transaktionen mit choreografierten Sagas



Ein großes Problem bei choreografierten Sagas ist es den Überblick über den aktuellen Stand zu behalten. Durch die Verwendung einer "Korrelations-ID" kann diese Problem gemindert werden.

Die Wahl der Softwarearchitektur ist immer eine Abwägung von vielen Tradeoffs!

47

Weitere Aspekte, die berücksichtigt werden können/müssen:

- Cloud (und ggf. Serverless)
- Mechanical Sympathy
- Testen und Deployment von Microservices (Stichwort: *Canary Releases*)
- Monitoring und Logging
- Service Meshes
- ...

Literatur

[[Newman2021](#)] Sam Newman, **Building Microservices: Designing Fine-Grained Systems**, O'Reilly, 2021.