Zusammenfassung

Verteilte Software Systeme

Michael Wieland and Fabian Hauser

Hochschule für Technik Rapperswil

9. Juli 2017

Mitmachen
Falls du an diesem Dokument mitarbeiten möchtest, kannst du es auf GitHub unter `https://github.com/michiwieland/hsr-zusammenfassungen` forken.

Inhaltsverzeichnis

1. Introduction

Distributes System: A distributed system is a collection of independent computers that appears to its users as a single coherent system.

1.1. Herausforderungen und Ziele VSS

Herausforderungen:

- Komplexe Kommunikation / Systeme
- Performanzprobleme
- Zuverlässigkeit
- Transaktionssicherheit

Ziele:

- Remote-Kommunikation heterogener Anwendungsteile (Interoperabilität)
- Hohe Benutzerzahlen (Scalability)
- Fault Tolerance (Robustness)

1.1.1. Häufige Fehlannahmen

Eight Fallacies of Distributed Systems (dt. Trugschlüsse, nicht zutreffende Annahmen, siehe http://www.rgoarchitects.com/Files/fallacies.pdf):

- The network is reliable.
- Latency is zero.
- Bandwidth is infinite.
- The network is secure.
- Topology doesn't change.
- There is one administrator.
- Transport cost is zero.
- The network is homogeneous.

Fowler's First Law of Distributed Object Design, siehe: http://www.drdobbs.com/errant-architectures/184414966

- Don't distribute your objects!

1.1.2. ACID

ACID ist eine Voraussetzung für die Verlässlichkeit von verteilten Systemen.

Atomicity Ganz oder gar nicht

Consistency Nach jeder Transaktion müssen die Daten konsistent sein

Isolation Nebenläufige Ausführungen beeinflussen sich nicht

Durability Änderungen an den Daten bleiben persistent

1.2. Middlewares

Üblicherweise wird ein verteiltes Software System über eine Middleware gesteuert. Gründe für die Einführung von Middleware:

- Überwindung von Heterogenität (Interoperabilität, Transparenz)
- Vereinfachte Erstellung verteilter Anwendungen (Produktivität)

Die Middleware kann folgende Funktionen umfassen: Kommunikation, Naming, Events, Security, Transactions etc.

1.2.1. Arten von Middlewares

Kommunkationsorienterte Middleware Abstraktion Netzwerkzugriff. Implementierungen: z.B. Sockets (low-level Protokolle)

Anwendungsorientierte Middleware Weitreichente Unterstützung verteilter Anwendung z.B. Discovery, Sicherheit, Zuverlässigkeit, verteilte Transaktionen, Sessions. Implementierungen: CORBA IDL, RMI interfaces, WSDL/SOAP Web Services (high-level Protokolle)

1.3. Architekturstile

Stil	Topologie-Muster	Verbreitung	Known Uses
Client-Server	n:1 (2-Tier, 3-Tier, n-Tier)	Mainstream	WWW (HTTP) Web Services Microservices DCE RPC
Distributed Objects	n:m	Nach Boom jetzt gering	CORBA, RMI
Hub-and-Spoke with Messaging	Stern, Bus (also n:1:m)	Mainstream	ESB
Event-Driven Architecture (EDA)	Wie Hub-and-Spoke	Forschungsthema Prototypen	IoT
Shared Data	n:1	Forschungsthema Prototypen	Data Spaces
Peer-to-Peer (P2P)	n:m, oft mit Client-Server als unterliegender Kommunikationsinfrastruktur)	Hoch in Nischen-Einsatzgebieten	File Sharing Music Sharing

Abbildung 1: Architekturen von VSS

1.3.1. Design Aspects

- Communication Topology (Clients, Network, Server Nodes)
  - One client, one server? Many clients for one server? Dynamic setup?
- Location Autonomy (Transparency)
  - Real address vs. virtual address (what if a server is moved?)
- Invocation and Message Semantics
  - Bytestream vs. Document vs. Procedure vs. Remote Object
- Timeout Management

  - 1ms? 30 seconds? Infinite?

- Error Handling
  - Retry request?
  - Make server idempotent?

1.4. Tiers

- Two-Tier (Aufteilung in Client und Server)
- Three-Tier (Aufteilung in Clients, Logic und Backend bzw. oft Client, Applikation und Datenbank)

1.5. Integrationen

- File Transfer
- shared Database (Common)
- Remote Procedure Invocation
- Messaging (Common exchange): Üblicherweise identifikation der Response über request ID

1.6. Transparency

Concept	Example
Centralized services	A single server for all users
Centralized data	A single on-line telephone book
Centralized algorithms	Doing routing based on complete information

Abbildung 2: Scalability Issues / Centralized Concept Services

Transparency	Description
Access	Hide differences in data representation and how a resource is accessed
Location	Hide where a resource is located
Migration	Hide that a resource may move to another location
Relocation	Hide that a resource may be moved to another location while in use
Replication	Hide that a resource is replicated
Concurrency	Hide that a resource may be shared by several competitive users
Failure	Hide the failure and recovery of a resource

Abbildung 3: Transparancy Types ISO 1995

1.7. Dimensionen der Schwierigkeit

Nebenläufigkeit gleichzeitig / concurrency

Persistenz/Transparenz Sichbarkeit der Verteilung / tatsächlichen Orten

Verteilung/Transparenz Sichbarkeit, ob eine Ressource Lokal/Verteilt ist

1.8. Inter Process Communication (IPC)

Die drei Varianten unterscheiden sich in Punkto Unterstützung von Sicherheit, Zuverlässigkeit, Transaktionsmanagement, Kopplung und QoS.

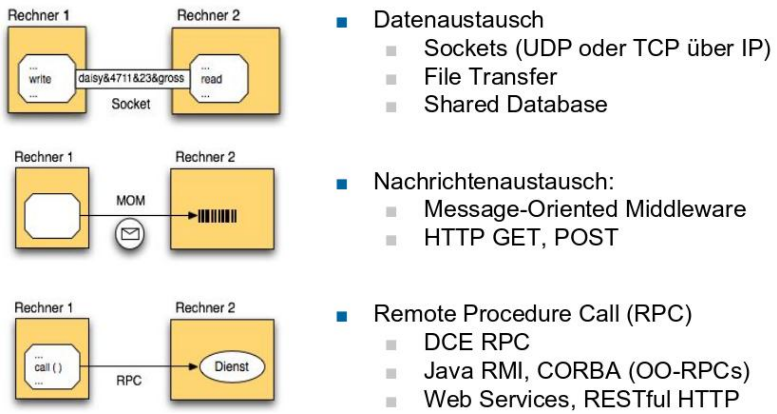

Abbildung 4: IPC Abstraktionsebenen

2. Technologien

2.1. Sockets

Basis Mechanismus für alle komplexere Verfahren (HTTP, RMI, etc.)

- Austausch von Bytetrömen
- Socket ist Verbindung zwischen zwei Kommunikationsendpunkten (IP/Port)
- Zwei Rollen: Client/Server

Vorteile:

- Flexibel und mächtig
- Effizient (hinsichtlich Network Traffic)
- Genügt z.B. um aktualisierte Informationen zu übermitteln

Nachteile:

- Nur übermitteln von Rohdaten
- Eigenes Konstruieren und Parsen von Bytetrömen
- Message Exchange Pattern (MEP) muss selbst spezifiziert, implementiert und überprüft werden. Das MEP definiert die Reihenfolge der `send()` und `receive()` Calls zwischen Client/Server.
- Client & Server müssen State-Informationen vorhalten.

2.1.1. TCP

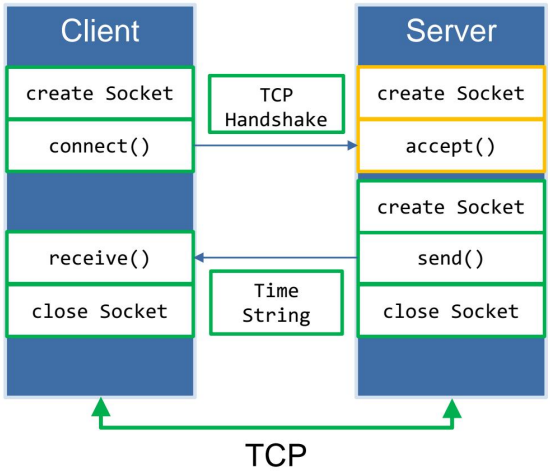

Abbildung 5: TCP Socket

Client

1. Find the IP address and port number of the server
2. Create a TCP Socket
3. Connect the Socket to the server (server must be up and listening for new requests)
4. Send data to / receive data from server using the socket
5. Close the connection

```
1  public class DaytimeTCPClient {
2
3     private final String host = "localhost";
4     private final int port = 5000;
5
6     private void connectToServer() throws Exception {
7        try (Socket server = new Socket(host, port)) {
8           BufferedReader in = new BufferedReader(new
                InputStreamReader(server.getInputStream()));
9           String date = in.readLine();
10          System.out.println(date);
11       }
12
13    }
14
15    public static final void main(String[] args) throws Exception {
16       DaytimeTCPClient client = new DaytimeTCPClient();
17       client.connectToServer();
18    }
19 }
```

Server

1. Find the IP address and port number of the server
2. Create a TCP server socket
3. Bind server socket to server IP and port number (this is the port to which clients will connect)
4. Accept new connection from client (returns client Socket that represents the client which is connected)
5. Send data to / receive data from client, using client socket
6. Close the connection with client

```
1  public class DaytimeTCPServer {
2
3     private final int port = 5000;
4
5     private void listen() throws Exception {
6        try (ServerSocket server = new ServerSocket(port)) {
7           while (true) {
8              Socket client = server.accept();
9
10             PrintWriter out = new PrintWriter(client.getOutputStream(), true);
11             Date date = new Date();
12             out.println(date);
13          }
14       }
15    }
16
17    public static void main(String[] args) throws Exception {
18       DaytimeTCPServer server = new DaytimeTCPServer();
19       server.listen();
20    }
21 }
```

2.1.2. UDP

Wird eingesetzt für Low-Latency, State-less Übertragungen, bei denen Paketverlust keine (schlimme Konsequenzen hat). Es gibt keine Garantie für die Datenübertragung, Reihenfolge und Atomizität der Pakete.

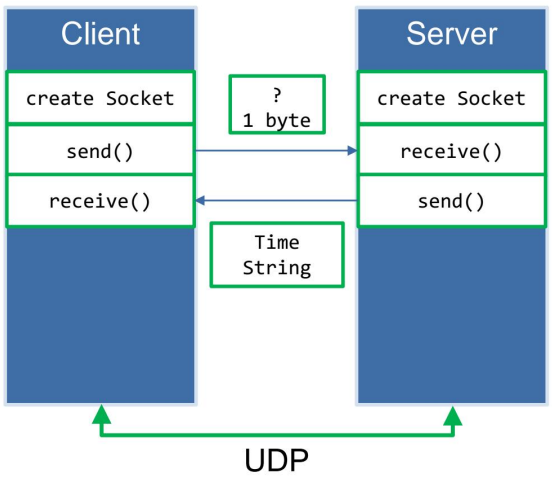

Abbildung 6: UDP Socket

Client

```
1  public class DaytimeUDPClient {
2
3     private final String targetHost = "localhost";
4     private final int targetPort = 5000;
5     private final int bufferLength = 256;
6
7     public DaytimeUDPClient() throws Exception {
8        try (DatagramSocket socket = new DatagramSocket()) {
9           // send control datagram
10          InetSocketAddress serverAddr = new InetSocketAddress(targetHost, targetPort);
11          byte[] controlData = {1};
12          DatagramPacket controlPackage = new DatagramPacket(controlData,
                controlData.length, serverAddr);
13
14          socket.send(controlPackage);
15
16          // receive answer
17          byte[] responseBuffer = new byte[bufferLength];
18          DatagramPacket response = new DatagramPacket(responseBuffer,
                responseBuffer.length);
19          socket.receive(response);
20
21          System.out.println(new String(responseBuffer, 0, response.getLength()));
22       }
23    }
24    public static void main(String[] args) throws Exception {
25       new DaytimeUDPClient();
26    }
27 }
```

Server

```
1  public class DaytimeUDPServer {
2
3     private final int targetPort = 5000;
4
5     public DaytimeUDPServer() throws Exception {
6
7        try (DatagramSocket socket = new DatagramSocket(targetPort)) {
8
9           while (true) {
10             // control frame
11             DatagramPacket packet = new DatagramPacket(new byte[1], 1);
12             socket.receive(packet);
13
14             // send response
15             String dateString = new Date().toString();
16             byte[] outBuffer = dateString.getBytes();
17             DatagramPacket date = new DatagramPacket(outBuffer, outBuffer.length,
                   packet.getSocketAddress());
18             socket.send(date);
19          }
20       }
21    }
22
23    public static void main(String[] args) throws Exception {
24       new DaytimeUDPServer();
25    }
26 }
```

2.1.3. Berkley Sockets

Ist ein Message Passing Interface

2.1.4. Websockets

Im Web meistens Server zu Client; aber neu in einigen Fällen synchrone Kommunikation nötig. Websockets werden von vielen Plattformen unterstützt.

3. Messaging Patterns

Strukturen, welche zur Datenübertragung genutzt werden:

3.1. Blocking Receiver Message Pattern

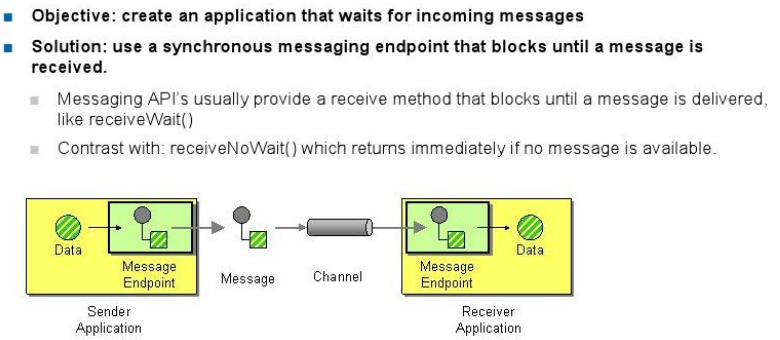

Abbildung 7: Blocking Receiver Message Pattern

3.2. Polling (Non-Blocking) Receiver Message Pattern

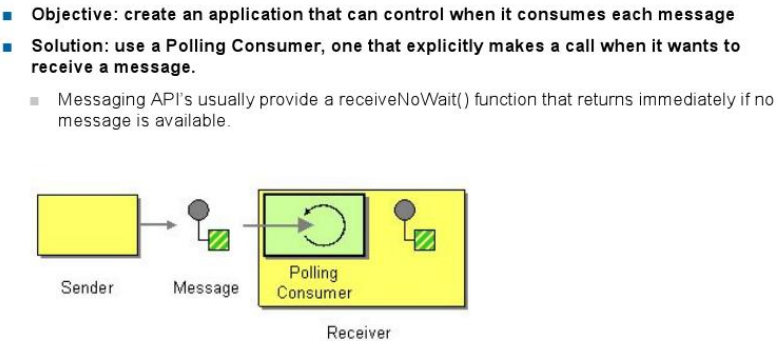

Abbildung 8: Polling (Non-Blocking) Receiver Message Pattern

3.3. Service Activator Message Pattern

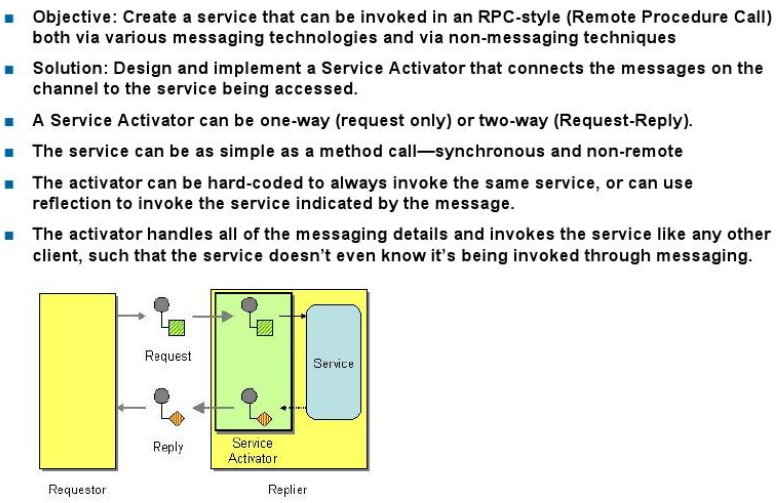

Abbildung 9: Service Activator Message Pattern

3.4. Message Exchange Patterns

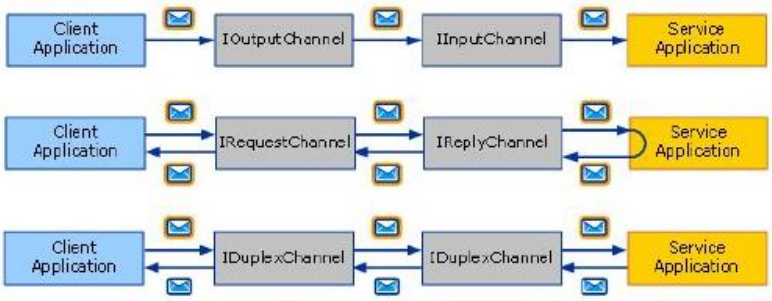

Abbildung 10: Message Exchange Patterns (as defined by WCF)

3.4.1. Exchanges and Exchange Types

Exchanges are AMQP (Advanced Message Queuing Protocol) entities where messages are sent. Exchanges take a message and route it into zero or more queues. There are four types of exchanges:

Direct exchange Delivery based on routing key

Fanout exchange Delivery to all the queues that are bound.

Topic exchange Route messages to one or many queues based on matching between a message routing key and the pattern that was used to bind a queue to an exchange.

Headers exchange Routing on multiple attributes that are more easily expressed as message headers than a routing key

4. Messaging

4.1. Übersicht

4.1.1. Merkmale

Messaging wird heute vielfach als einfacherer Ansatz für die Integration unterschiedlicher Systeme eingesetzt mit den Merkmalen:

- Einfachheit
- Lose Kopplung
- Erweiterbarkeit
- Skalierbarkeit
- Fehlertoleranz

4.1.2. Queue-basiertes Messaging

Gestattet die flexible und lose Kopplung unterschiedlichster Systeme:

- Auf unterschiedlichen Plattformen
- In unterschiedlichen Programmiersprachen
- Mit völlig unterschiedlichen Message-Formaten (Text, Byte, Objekt)

4.1.3. Enterprise Integration Patterns

Der Nachrichtenaustausch erfolgt in der Regel mittels eines der zwei Channel-Typen:

- Point-to-Point (P2P) liefert jede Nachricht an genau einen Empfänger
- Publish-Subscribe kopiert Nachrichten (Lieferung an mehrere Empfänger)

4.1.4. Designentscheide

Die APIs sind einfach zu benutzen, es müssen aber viele Designentscheidungen getroffen werden:

- Message intent (command vs. data)
- Returning a response (request-reply)
- Huge amounts of data (sequencing)
- Slow messages (message expiration)
- QoS (guaranteed delivery, transactionality, idempotency)

4.2. Message

A message is a container that consists of:

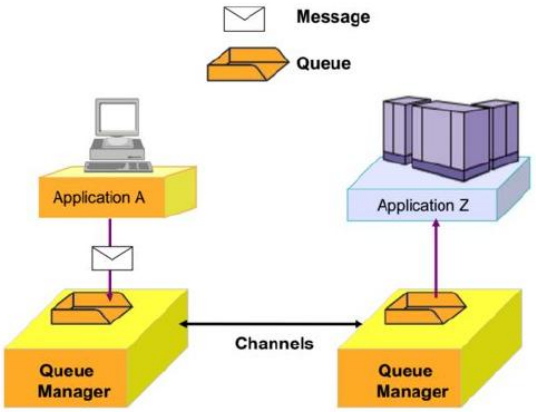

Abbildung 11: Messaging Queue manager

Message Descriptor: Identifies the message and contains control information (e.g. message type, and priority) that is assigned to the message by the sending application.

Message data: Contains the application data. The structure of the data is defined by the application programs that use it, and the queue manager is largely unconcerned with its format or content.

Queue Manager Is responsible for accepting and delivering messages. Maintains queues of all messages that are waiting to be processed or routed.

Message Queuing Model

Sender und Empfänger einer Queue können diese entweder aktiv oder passiv abfragen bzw. abfüllen.

4.3. Two Phase Commit (2PC)

2PC garantiert, dass alle Teilnehmer ihre Aktion auch wirklich durchführen können.

1. **Commit-Request Phase** Koordinator forder alle Teilnehmer auf, die Transaktion probeweise durchzuführen. Die Teilnehmer antworten mit READY oder FAILED.

2. **Commit Phase** Wenn alle Teilnehmer die Transaktion durchführen können (READY) , gibt der Koordinator die Aufforderung die Transaktion wirklich auszuführen. (COMMIT)

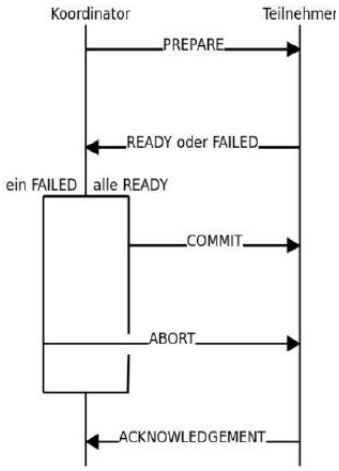

Abbildung 12: Two phase Commit (2PC)

4.4. Java Message Service (JMS)

Der JMS ist ein Programmierinterface, welches es erlaubt, mit einem Queue Manager zu sprechen.

4.4.1. Reliability Levels:

Nachrichten können sowohl persistent als auch nicht-persistent verwaltet werden; es gibt weitere Quality of Service (QoS)-Properties.

Best effort nonpersistent Messages are discarded when a messaging engine stops or fails. Messages might also be discarded if a connection used to send them becomes unavailable or as a result of constrained system resources.

Express nonpersistent Messages are discarded when a messaging engine stops or fails. Messages might also be discarded if a connection used to send them becomes unavailable.

Reliable nonpersistent Messages are discarded when a messaging engine stops or fails.

Reliable persistent Messages might be discarded when a messaging engine fails.

Assured persistent Messages are not discarded.

Beispiel Sender:

```
1  Connection connection; connection = connectionFactory.createConnection();
2  connection.start();
3  Session session = connection.createSession(mp.transacted, Session.AUTO_ACKNOWLEDGE);
4  Destination destination = session.createQueue("testQueue");
5  MessageProducer producer = session.createProducer(destination);
6  producer.setDeliveryMode(DeliveryMode.PERSISTENT);
7  TextMessage message = session.createTextMessage("hello");
8  producer.send(message);
9  producer.close();
10 session.close();
11 connection.close();
```

Beispiel Receiver/Consumer:

```
1  Connection connection = connectionFactory.createConnection();
2  connection.start();
3  Session session = connection.createSession(mc.transacted, Session.AUTO_ACKNOWLEDGE);
4  Destination destination = session.createQueue("testQueue");
5  MessageConsumer consumer = session.createConsumer(destination);
6  TextMessage text=(TextMessage) consumer.receive();
7  consumer.close();
8  session.close();
9  connection.close();
```

4.5. Point-to-Point Pattern

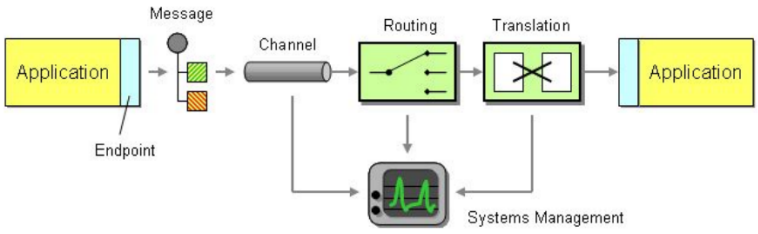

Abbildung 13: Point-to-Point Messaging Pattern

Beim P2P Pattern wird eine Message in 5 Schritte übertragen:

Create the sender creates the message and populates it with data.

Send the sender adds the message to a channel.

Deliver the messaging system moves the message from the sender's computer to the receiver's computer, making it available to the receiver.

Receive the receiver reads the message from the channel.

Process the receiver extracts the data from the message

Message Expiration Falls der Empfänger nicht verfügbar ist, häuffen sich die Messages in der Queue an. Es empfiehlt sich deshalb, eine Expiration Wert anzugeben, damit die Messages nach einer bestimmten Zeit aus der Queue entfernt werden.

Document Message Document Messages werden verwendet um Datenstrukturen auf eine verlässliche Art und Weise zwischen Sender und Empfänger zu übertragen.

Command Message Command Messages werden verwendet um Befehle an den Empfänger zu übertragen, die dieser ausführen soll.

4.6. Publish-Subscribe Pattern

Beim Publish-Subscribe wird die Message an einen Publish-Subscribe Channel gesendet, welche eine Kopie der Meldung an alle Teilnehmer weiterleitet. (Observer)

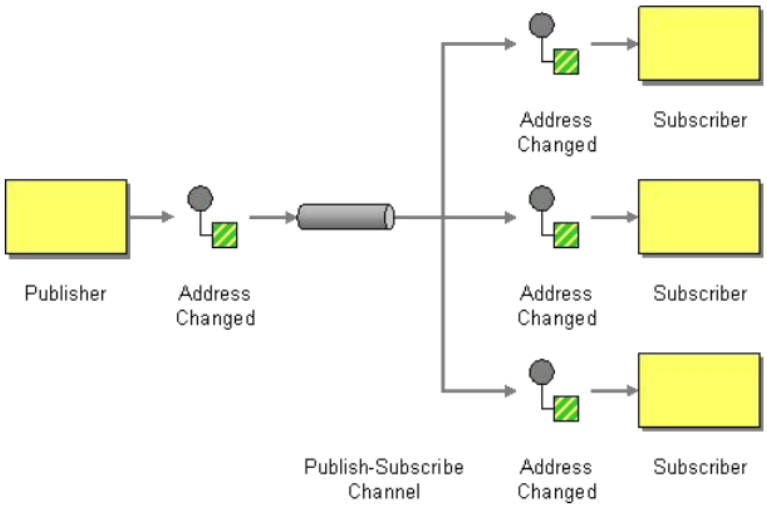

Abbildung 14: Publish Subscribe Integration Pattern

Man unterscheidet zwischen zwei Typen:

Topic-based Der Publisher beschreibt ein Thema/Typ und der Subscriber interessiert sich für ein oder mehrere Thema/Typen. Die Subscriber erhalten nur noch Meldungen die sich auch interessieren.

Content-based Der Subscriber definiert an welchem Inhalt er genau interessiert ist. Es gibt dabei spezifische Attribute an, welche erfüllt sein müssen.

4.6.1. Publisher JMS

```
1  public void run() {
2     try {
3        HelloWorldPublisher mp = new HelloWorldPublisher();
4        ActiveMQConnectionFactory connectionFactory = new
             ActiveMQConnectionFactory(mp.user, mp.password, mp.url);
5        Connection connection = connectionFactory.createConnection();
6        connection.start();
7        Session session = connection.createSession(mp.transacted,
             Session.AUTO_ACKNOWLEDGE);
8        Destination destination = session.createTopic(mp.topicName);
9        MessageProducer producer = session.createProducer(destination);
10       TextMessage message = session.createTextMessage(mp.messageText);
11       producer.send(message);
12    } catch (JMSException e) {
13    e.printStackTrace();
14    } catch (Exception e1) {
15    e1.printStackTrace();
16    }
17 }
```

4.6.2. Subscriber JMS

```
1  public static void main(String[] args) {
2     try {
3        HelloSubscriber hs = new HelloSubscriber();
4        ActiveMQConnectionFactory connectionFactory = new
             ActiveMQConnectionFactory(hs.user, hs.password, hs.url);
5        Connection connection = connectionFactory.createConnection();
6        Session session = connection.createSession(hs.transacted,
             Session.AUTO_ACKNOWLEDGE);
7        Destination destination=session.createTopic(hs.topicName);
8        MessageConsumer consumer= session.createConsumer(destination);
9        connection.start();
10       TextMessage text = (TextMessage) consumer.receive();
11    } catch (Exception e1) {
12       e1.printStackTrace();
13    }
14 }
```

4.7. RabbitMQ

Abbildung 15: RabbitMQ Pattern

5. Remote Procedure Call (RPC)

Ein Remote Procedure Call verläuft analog zu einem lokalen Funktionsaufruf, jedoch auf einem entfernten Server. Mit RPC lassen sich verteilte, Client-Server Applikationen erstellen.

- Der Entwickler muss sich bewusst sein, dass ein RPC wegen Netzwerkprobleme fehlschlagen kann. Der Aufrufer muss mit solchen Fehler umgehen, ohne genau zu wissen, ob die Prozedur beim Server jemals aufgerufen wurde. (Timeout Management)
- Nicht idempotente Methoden sind oft keine guten Kanditaten für RPC.
- RMI hat eine hohe Kopplung zwischen Client und Server

5.1. Java Remote Method Invocation (RMI)

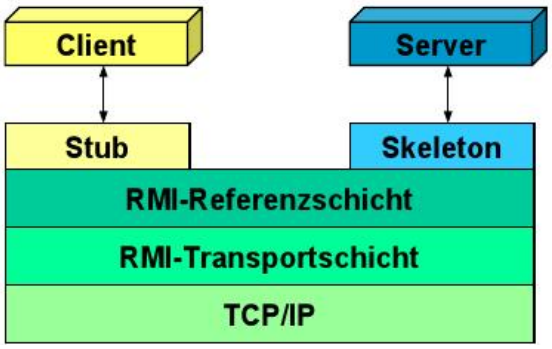

Abbildung 16: RMI Stack

- Ein Stub ist ein Stellvertreterobjekt (Remote Proxy), das Clientaufruf an Server weiterreicht. Der Procedure Call wird als Message an das OS weitergereicht, welches die Meldung an den Server sendet.
- Ein Skeleton nimmt Aufrufe des Stubs entgegen und leitet sie an das Serverobjekt weiter. Solange der Server arbeitet ist der Client typischerweise blockiert. (synchron) Das Serverobjekt (Remote Object) beinhaltet die Methoden, welche aufgerufen werden sollen, sowie einen Status.
- Stub und Skeleton implementieren das selbe Interface.
- RMI-Referenzschicht stellt Namensdienst (=Registry) zur Verfügung
- RMI-Transportschicht verwaltet Verbindungen und wickelt Kommunikation ab

5.1.1. Ablauf

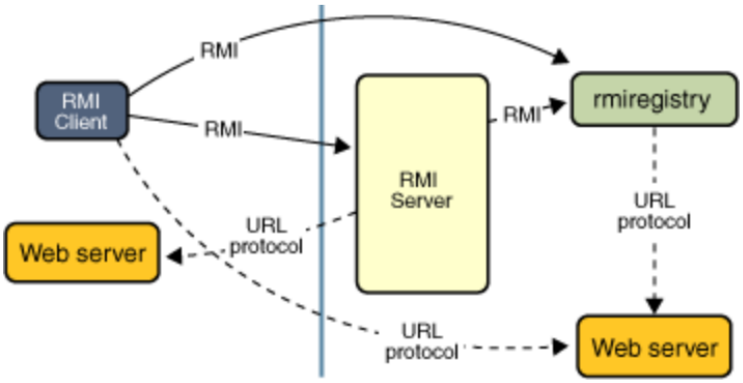

Abbildung 17: RMI Registry

1. Start RMI Registry
2. Start Server
3. The server calls the registry to associate (or bind) a name with a remote object.
4. Start the client
5. The client looks up the remote object by its name in the server's registry and then invokes a method on it.
6. The illustration also shows that the RMI system uses an existing web server to load class definitions, from server to client and from client to server, for objects when needed.
7. Call Methods within remote object

5.1.2. Implementation

RMI Klassenimplementationen können vom Server zu den Clients und umgekehrt übertragen werden. Somit können Änderungen an der Businesslogik einmalig beim Server implementiert und an die Clients verteilt werden.

Stub-Klasse in RMI (Remote Proxy)

Die Stub-Klasse baut Socket-Verbindung zu Server auf (connect-Call). Sie schickt Namen der Methode und Parameter und holt das Ergebnis ab.

```
1  public class HelloImpl_Stub implements Hello {
2     Socket socket = new Socket("[hostname]",4711);
3     ObjectOutputStream outStream = new ObjectOutputStream(socket.getOutputStream());
4     // fuer remote Methode sayHello():
5     public String sayHello( ) throws RemoteException{
6        outStream.writeObject("sayHello"); // simplified example
7        ObjectInputStream in = new ObjectInputStream(socket.getInputStream());
8        return (String)in.readObject();
9     }
10 }
```

Skeleton-Klasse in RMI

Die Skeleton Klasse erzeugt einen ServerSocket und wartet auf Methoden-Aufrufe von Client und delegiert diesen an das Remote Objekt. Danach liefert sie den Rückgabewert über die Socketverbindung (Port) an den Client zurück.

```
1  public class HelloImpl_Skeleton extends Thread{
2     HelloImpl myHello;
3     ObjectInputStream inStream = new ObjectInputStream(socket.getInputStream());
4     public void run(){
5        ServerSocket serverSocket = new ServerSocket(4711);
6        Socket socket = serverSocket.accept();
7        String method = (String) inStream.readObject(); // simplified ex.
8        if(method.equals("sayHello")){
9           String return = myHello.sayHello();
10          ObjectOutputStream outStream = new
                ObjectOutputStream(socket.getOutputStream());
11          outStream.writeInt(return);
12       }
13    }
14 }
```

RMI-Programmierung Serverseite

```
1  // Interface fuer Remote-Methoden (fuer Client und Server)
2  import java.rmi.Remote;
3  import java.rmi.RemoteException;
4  public interface Hello extends Remote {
5     String sayHello() throws RemoteException;
6  }
7
8  // Klasse zur Implementierung des Remote-Interface
9  import java.rmi.RemoteException;
10 import java.rmi.server.UnicastRemoteObject;
11 public class HelloImpl extends UnicastRemoteObject implements Hello {
12    public HelloImpl() throws RemoteException {
13       super();
14    }
15    public String sayHello() throws RemoteException{
16       return "Hello World!";
17    }
18 }
19
20 // remote-Objekte erzeugen und in RMI-Registry anmelden
21 import java.rmi.Naming;
22 public class HelloServer {
23    public static void main(String args[]) {
24       try {
25          HelloImpl obj = new HelloImpl();
26          Naming.rebind("rmi://[hn]/remoteHello", obj);
27       }catch (Exception e) { ... }
28    }
29 }
```

5.2. Interface Definition Language (IDL)

Interface Definition Language (IDL) ist eine plattform-unabhängige Sprache zur Definition von RPC-Interfaces.

5.3. Web Service Description Language (WSDL)

Web Services Description Language (WSDL) ist eine XML-Basierte IDL, die beim W3C spezifiziert ist. Es ist gut standardisiert (und funktioniert daher auch in mehreren Programmiersprachen), hat allerdings durch XML einen eher grossen Overhead. (4-20x grösser wie der Payload) Man unterscheidet zwischen zwei WSDL Styles, wobei der RPC-Style eine festgelegte Struktur hat, der Document-Style dagegen keine Vorgaben definiert.

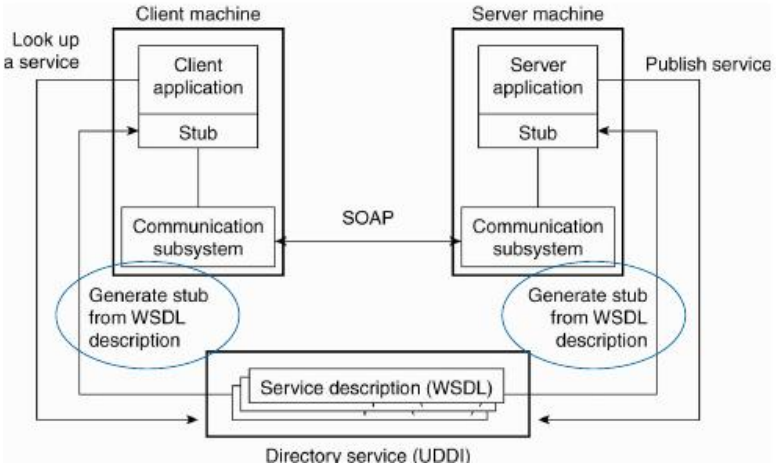

Abbildung 18: WSDL Übersicht

5.4. Flat Nameing

5.4.1. ARP

5.5. Structural Nameing

Hat eine üblicherweise Hierarchie.

5.5.1. DNS

Iterative resolution Directly ask all nameservers partially.

Recursive resolution Ask another nameserver to resolve a domain name.

5.6. Attribute-Based Naming

Attribute-Based Nameing usually has a Directory Services e.g. LDAP or AD
The main challange is to find a sensible set of attributes (key, value) for all users.

5.6.1. Bonjour / Zeroconf

Bonjour / Zeroconf allows automatic name resolving in a local network. It combines DHCP, mDNS (multicast DNS) and SDP (Serice Discovery Protocol). It's aimed at small home networks and only works locally as all parties must be trusted.

6. Distributed Data Structures

6.1. Distributed Hash Tables

Usually, to Distribute e.g. files over a number n of nodes, one could use a hash function: $location(fileURL) = hashCode(fileURL)\ \%\ n$. The problem here is that if n changes, the system has to re-distribute all files > the old n.

6.1.1. Chord

Namespace Namespache is 2^m with typically $m = 128$ or $m = 160$. Nodes are considered to be in a Modulo Ring formation. Each node knows, which node is its successor.

Storing Resources To store ressources by keys, the storage space is devided up: each node stores keys smaller than its own ID up to the predecessor node.

The $succ()$ function is distributed and is used as lookup function. Every host has a $successor$ value. If the system is stable, $p.successor = succ(p + 1)$.

m m-Bit Identifiers $\rightarrow 2^m$ Namespace size

p ID of a node: p = hash(IP)

k Key of a Ressource

q Nearest predecessor in finger table

When joining, the new node lands in a addressspace of a existing node. The predecessors and successors must be convinced that the new node is responsible for its part of the namespace. To allow easy joining, every node also knows his predecessor.

When leaving, the storage space is transferred to the node's neightbours. The node then leaves the ring.

The finger table allows faster access ($\mathcal{O}(log \cdot N)$) to more nodes than only the predecessor and successor:

$p.finger[i]$ points to $suc(p + 2^{i-1})$

$p.finger[1] = succ(p + 1) = p.successor$

Lookup

1. Am I responsible? (is k between me and predecessor) Return my ID to requestor.
2. Otherwise: Is my successor responsible? Forward request to successor
3. Otherwise: Forward request clockwisely to predecessor

7. Synchronization / Logical Clocks

7.1. Physical Clocks

7.2. Lamport Logical Clock

8. Bitcoin

A. Listings

B. Abbildungsverzeichnis

C. Tabellenverzeichnis