Verteilte Systeme - Zusammenfassung

Florian Thiévent

4. Semester (FS 2019)

Inhaltsverzeichnis

1 Networking

1.1 InetAddress

Static factory methods

- getByName(String name)
- getByAddress (4/16 bytes)
- getAllByName(String host)
- getLocalHost()

Instance methods

- byte[] getAddress()
- String getHostAddress()
- String getHostName()
- String getCanonicalHostName()
- boolean isReachable(int timeout)
- boolean isMulticastAddress()

1.2 Network Interfaces

Listing 1: Network Interfaces and its addresses

```
public static void main(String[] args) throws SocketException {
  Enumeration<NetworkInterface> interfaces = NetworkInterface.getNetworkInterfaces();
  while(interfaces.hasMoreElements()){
    NetworkInterface intf = interfaces.nextElement();
    Syst  em.out.print(intf.getName());
    System.out.println(" ["+intf.getDisplayName()+"]");
    Enumeration<InetAddress> adr = intf.getInetAddresses();
    while(adr.hasMoreElements()){
      System.out.println("\t" + adr.nextElement());
    }
    byte[] hardwareAddress = intf.getHardwareAddress();
  }
}
```

1.3 Sockets

Abstraction through which an application may send and receive data through the network. A Socket is identified by Hostname/IP and port number.

Stream Sockets

- Use TCP as end-to-end protocol
- Provide a reliable byte-stream
- Connection oriented: Socket represents one end of a TCP connection

Datagaram Sockets

- Use UDP as protocol
- Not connection oriented, not reliable

1.3.1 Controlling Socket Behaviors

Blocking & Timeouts

ServerSocket.accept / InputStream.read
read or accept call will not block for more than a fixed number of msec otherwise, InterruptedIOException is thrown (get/setSoTimeout(int timeout))

Socket constructor
Uses a system-defined timeout, cannot be changed by Java API (Solution: use connect)

OutputStream.write
Cannot be interrupted / caused to time-out by Java API

Keep-Alive

- TCP provides a keep-alive mechanism
- Probe messages are sent after a certain time
- Application only sees keep-alive working if the probes fail!
- Per default keep-alive is disabled
- Default timeout: 2h (7200 secs)

Send / Receive Buffer Size

- When a Socket is created, the OS must allocate buffers to hold incoming & outgoing data
- Receive buffer size may also be specified on server socket (for accepted sockets which immediately receive data)

No Delay

- TCP tries to avoid sending small packets
- Buffers data until it has more to send, combines small packets with larger ones
- Necessary if application has to be efficient
- Default: false

1.3.2 Closing Connections

close()

- Once an endpoint (client or server) closes the socket, it can no longer send or receive data
- Close can only be used to signal the other end that the caller is completely finished communicating

shutdownOutput()

- Closes output-stream, no more data can be may be written (IOException)
- All data written before shutdownOutput can be read by receiver

shutdownInput()

- Closes the input stream
- Any undelivered data is (silently) discarded, read operations will return -1

s.close() / s.shutdownOutput()

- Data may still be waiting to be delivered to the other side
- By default, socket tries to deliver remaining data, but if socket crashes, data may be lost without notification to sender (as close returns immediately)

1.3.3 User Datagram Protocol

- UDP allows to address applications over ports
- UDP adds another layer of addressing (ports) to that of IP
- UDP detects some form of data corruption that may occur in transit and discards corrupted messages
- UDP retains message boundaries

2 Internet

2.1 Protocol

GET

- Access of content from the server
- Idempotent, i.e. the side effects of N¿0 identical requests is the same as for a single request (f(f(x)) = f(x))

POST
Comparable to GET but Method must not necessarily be idempotent and Request data is transferred in the body of the request

HEAD

- Identical to GET, except that the server must not return the body
- Can be used to request meta information (headers) about the resource

OPTIONS (1.1)
Returns information about the communication options available on the specified resource (or on the server in general if request URI=*)

PUT (1.1)
Stores a web page on the server (rarely implemented)

DELETE (1.1)
Removes a web resource from the servver (rarely implemented)

TRACE (1.1)
Returns the request as it was accepted by server (⇒ debugging)

CONNECT (1.1)
Implemented by Proxy Server capable to provide an SSL tunnel

2.1.1 Response Codes

200-299: Success

200 OK

201 Created

202 Accepted

300-399: Redirections

300 Multiple Choices

301 Moved Permanently

302 Found

303 See Other (e.g. after POST)

304 Not Modified

305 Use Proxy

307 Temporary Redirect

400-499: Client Error

400 Bad Request

401 Unauthorized

402 Payment Required

403 Forbidden

404 Not Found

405 Method Not Allowed

407 Proxy Authentication Required

408 Request Time-out

411 Length Required

413 Request Entity Too Large

414 Request-URI Too Large

415 Unsupported Media Type

500-599: Server Error

500 Internal Server Error

501 Not Implemented

503 Service Unavailable

505 HTTP Version not supported

2.2 Request Headers

Host server host

Referer host from which the request is initiated

Accept data types supported by the client

Accept-Language language supported by client

Accept-Encoding encodings supported by client, e.g. gzip or deflate

User-Agent browser details, supplies server with information about the type of browser making the request

Connection: Keep-Alive browser is requesting the use of persistent TCP connections

2.3 Response Headers

Content-Type MIME-Type of content

Content-Length size of body (in bytes)

Content-Encoding compression algorithms

Location used by redirections

Date timestamp when the response was created

Last-Modified modification date of resource (assumed by server)

Expires date after which the result is considered stale

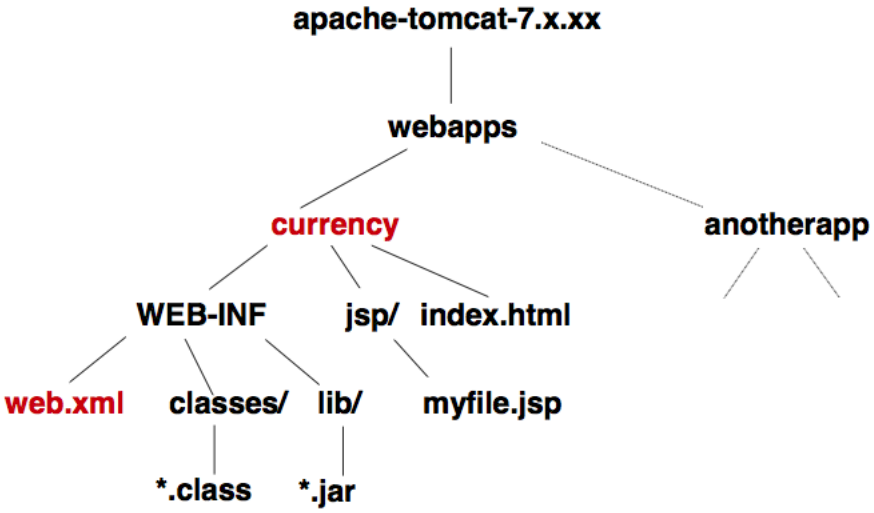
Server information about the server

Transfer-Encoding specifies type of transformation

Cache-Control information about cache handling (e.g. no-cache disables caching)

WWW-Authenticate information about authentication method

2.4 Servlet



Listing 2: Servlet Example

```
public class Converter extends HttpServlet {
  public void doGet(HttpServletRequest request, HttpServletResponse response) throws
      IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String amount = request.getParameter("amt");
    String from = request.getParameter("from");
    String to = request.getParameter("to");
    String res = computeResult(amount, from, to);
    out.println("<html>\n<body bgcolor=\"white\">");
    out.println("<h1>Currency Converter</h1>");
    out.println(amount + " " + from + " = " + res);
    out.println("</body>\n</html>");
  }
  String computeResult(String amount, String from, String to){...}
}
```

Listing 3: web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/
    XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.
    sun.com/xml/ns/javaee/web-app_3_0.xsd" version="3.0">
    <servlet>
        <servlet-name>CurrencyConverter</servlet-name>
        <servlet-class>ch.fhnw.ds.Converter</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>CurrencyConverter</servlet-name>
    <url-pattern>/convert</url-pattern>
  </servlet-mapping>
</web-app>
```

3 Webservices

3.1 XML-RPC

simples RPC Protokoll über HTTP, benötigt keine lange Einarbeitungszeit

3.1.1 Primitive Datentypen

- int, i4 — signed 32bit Integer
- string — ASCII string (no latin1)
- boolean — either 0 or 1
- double — double-precision floating point number
- dateTime.iso8601 — z.B. 20050717T14:08:14
- base64 — raw binary data, base64 encoded

Listing 4: Beispiele

```
<i4>13</i4>
<boolean>0</boolean>
```

3.1.2 Structs

- Struct enthält Members mit Name und Wert.
- können rekursiv sein (Structs die Structs enthalten)

Listing 5: Struct Beispiel

```
<i4>13</i4>
<struct>
  <member>
    <name>from</name>
    <value><i4>-5</i4></value>
  </member>
  <member>
    <name>to</name>
    <value><i4>5</i4></value>
  </member>
</struct>
```

3.1.3 Arrays

- Element-Typen können gemischt werden

Listing 6: Array

```
<array>
 <data>
  <value><i4>-5</i4></value>
  <value><string>44</string></value>
  <value><boolean>1</boolean></value>
 </data>
</array>
```

3.1.4 XML-RPC Request

Listing 7: Method Call

```
<?xml version="1.0" encoding="UTF-8"?>
<methodCall>
  <methodName>Echo.getEcho</methodName>
    <params>
      <param>
        <value>World</value>
      </param>
    </params>
</methodCall>
```

3.1.5 XML-RPC Response

Listing 8: Single Result

```
<?xml version="1.0" encoding="UTF-8"?>
<methodResponse>
  <params>
    <param>
      <value>Hello World, welcome to XML-RPC</value>
    </param>
  </params>
</methodResponse>
```

Als Resultat kann nur ein Wert zurückkommen, dieser kann jedoch auch ein Struct oder ein Array sein.

Listing 9: Fault Result

```
<?xml version="1.0" encoding="UTF-8"?>
<methodResponse>
  <fault>
    <value>
      <struct>
        <member>
          <name>faultCode</name>
          <value><i4>0</i4></value>
        </member>
        <member>
          <name>faultString</name>
          <value>No such handler: Echo.foo</value>
        </member>
      </struct>
    </value>
  </fault>
</methodResponse>
```

3.1.6 Apache XML-RPC Sample Server

Listing 10: Sample Server

```
import org.apache.xmlrpc.server.*;
import org.apache.xmlrpc.webserver.WebServer;

public class HelloServer {
  public static void main (String [] args) throws Exception {
    PropertyHandlerMapping phm = new PropertyHandlerMapping();
    phm.addHandler("Echo", ch.fhnw.ds.xmlrcp.echo.EchoImpl.class);
    WebServer server = new WebServer(80);
    XmlRpcServer xmlRpcServer = server.getXmlRpcServer();
    xmlRpcServer.setHandlerMapping(phm);
    server.start();
    System.out.println("Server started at port 80");
```

```
13    }
    }
```

Listing 11: Handler Class Server

```
1 public class EchoImpl {
  public String getEcho(String name) {
  return "[XML-RPC] Hello "+name+", welcome to XML-RPC";
  }
  }
```

Nur Instanzmethoden der Handlerklasse sind zugreifbar. Keine void Methoden. Public Default Constructor zwingend.

3.1.7 Apache XML-RPC Client

Listing 12: Handler Class Server

```
  import java.util.*;
  import org.apache.xmlrpc.*;

  public class HelloClient {
5   public static void main (String [] args) throws Exception {
      XmlRpcClientConfigImpl config = new XmlRpcClientConfigImpl();
      config.setServerURL(new URL("http://localhost/xmlrpc"));
      XmlRpcClient client = new XmlRpcClient();
      client.setConfig(config);
10    List params = new ArrayList();
      params.add(args[0]);
      Object result = client.execute("Echo.getEcho", params );
      System.out.println("The result is: "+result.toString());
    }
15 }
```

3.2 SOAP

3.2.1 WSDL (Web Services Description Language)

(früher Web Services Definition Language)
WSDL ist eine XML basierte Interface Beschreibungssprache die genutzt wird um die Funktionalität von Webservices zu beschreiben. Eine WSDL Beschreibung eines Webservices enthält:

- Wie der Service aufgerufen werden kann
- Welche Parameter er erwartet
- Welche Datenstruktur er zurückgibt

3.2.2 JAX-WS (Java API for XML Web Services

JAX-WS ist eine Java API um Webservices zu erstellen. Es ist Teil der Java EE (Enterprise Edition) Plattform von Sun Microsystems. Wie auch andere Java EE APIs nutzt JAX-WS Annotationen (@Webservice, @WebMethod usw). Basiert auf SOAP. Nur WSDL 1.1 unterstützt.

3.2.3 Anleitung JAX-WS

1. Interface erstellen

Listing 13: Interface erstellen

```
package ch.fhnw.ds.jaxws.server;
import javax.jws.WebService;
@WebService
public interface HelloService {
  String sayHello(@WebParam(name = "name") String name);
}
```

2. Interface implementieren

Listing 14: Interface implementieren

```
package ch.fhnw.ds.jaxws.server;
import java.util.Date;
@WebService
public class HelloServiceImpl implements HelloService {
  @Override
  public String sayHello(@WebParam(name = "name") String name){
    return "Hello " + name + " from SOAP at " + new Date();
  }
}
```

3. Java Objekte für XML Requests & Responses generieren % wsgen -cp bin -keep -s src -d bin ch.fhnw.ds.jaxws.server.HelloSer

- cp ¡path¿ classpath
- keep keep generated files
- s ¡path¿ path where to place generated source files
- d ¡path¿ path where to place generated output files
- ¡SEI¿ specify a SIB (service implementation bean)

Listing 15: SayHello

```
package ch.fhnw.ds.jaxws.server.jaxws;
import ...;
@XmlRootElement(name = "sayHello",
    namespace = "http://server.jaxws.ds.fhnw.ch/")
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "sayHello",
    namespace = "http://server.jaxws.ds.fhnw.ch/")
public class SayHello {

  @XmlElement(name = "name", namespace = "")
  private String name;
  public String getName() { return this.name; }
  public void setName(String name) { this.name = name; }
}
```

Listing 16: SayHelloResponse

```
package ch.fhnw.ds.jaxws.server.jaxws;
import ...;
@XmlRootElement(name = "sayHelloResponse",
    namespace = "http://server.jaxws.ds.fhnw.ch/")
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "sayHelloResponse",
    namespace = "http://server.jaxws.ds.fhnw.ch/")
public class SayHelloResponse {

  @XmlElement(name = "return", namespace = "")
  private String _return;
  public String getReturn() { return this._return; }
  public void setReturn(String _return) {
    this._return = _return;
```

```
15    }
  }
```

4. Service publishen

Listing 17: HelloServicePublisher

```
  package ch.fhnw.ds.jaxws.server;
  import javax.xml.ws.Endpoint;

4 public class HelloServicePublisher {
    public static void main(String[] args){
      Endpoint.publish(
        "http://127.0.0.1:9876/hs", // publication URI
        new HelloServiceImpl()); // SIB instance
9     System.out.println("service published");
    }
  }
```

5. Generierte Webservice Definition anschauen http://localhost:9876/hs?wsdl

Listing 18: WSDL

```
  <?xml version="1.0" encoding="UTF-8"?> <definitions   xmlns:soap="http://schemas.xmlsoap.
      org/wsdl/soap/"  xmlns:tns="http://server.jaxws.ds.fhnw.ch/"   xmlns:xsd="http://www.w3
      .org/2001/XMLSchema"  xmlns="http://schemas.xmlsoap.org/wsdl/"  targetNamespace="http
      ://server.jaxws.ds.fhnw.ch/"   name="HelloServiceImplService">
  <types>
    <xsd:schema>
4     <xsd:import namespace="http://server.jaxws.ds.fhnw.ch/" schemaLocation="http://
          localhost:9876/hs?xsd=1">
      </xsd:import>
    </xsd:schema>
  </types>
  <message name="sayHello">
9 <part name="parameters"
  element="tns:sayHello">
  </part>
  </message>
  <message name="sayHelloResponse">
14 <part name="parameters"
  element="tns:sayHelloResponse">
  </part>
  </message>
  <portType name="HelloServiceImpl">
19 <operation name="sayHello">
  <input message="tns:sayHello"></input>
  <output message="tns:sayHelloResponse"></output>
  </operation>
  </portType>
24 <binding name="HelloServiceImplPortBinding"
  type="tns:HelloServiceImpl">
  <soap:binding
  transport="http://schemas.xmlsoap.org/soap/http"
  style="document">
29 </soap:binding>
  <operation name="sayHello">
  <soap:operation soapAction=""></soap:operation>
  <input>
  <soap:body use="literal"></soap:body>
34 </input>
  <output>
  <soap:body use="literal"></soap:body>
  </output>
  </operation>
39 </binding>
  <service name="HelloServiceImplService">
  <port name="HelloServiceImplPort"
  binding="tns:HelloServiceImplPortBinding">
```

```
  <soap:address location="http://localhost:9876/hs">
44 </soap:address>
  </port>
  </service>
  </definitions>

49 Referenzierte Schema Definition:

  <?xml version="1.0" encoding="UTF-8"?>
  <xs:schema xmlns:tns="http://server.jaxws.ds.fhnw.ch/" xmlns:xs="http://www.w3.org/2001/
      XMLSchema"
  version="1.0"
54 targetNamespace="http://server.jaxws.ds.fhnw.ch/">
  <xs:element name="sayHello" type="tns:sayHello"></xs:element> <xs:element name="
      sayHelloResponse" type="tns:sayHelloResponse"></xs:element>
  <xs:complexType name="sayHello">
    <xs:sequence>
      <xs:element name="name" type="xs:string" minOccurs="0"></xs:element>
59  </xs:sequence>
  </xs:complexType>
  <xs:complexType name="sayHelloResponse">
    <xs:sequence>
      <xs:element name="return" type="xs:string" minOccurs="0"></xs:element>
64  </xs:sequence>
   </xs:complexType>
  </xs:schema>
```

6. Client Proxy generieren % wsimport -keep -p ch.fhnw.ds.jaxws.client.jaxws -d bin -s src

- -keep keep generated files
- -p ¡package¿ specify (overwrite) target package
- -s ¡path¿ path where to place generated source files
- -d ¡path¿ path where to place generated output files
- ¡WSDL¿ Web Service Definition
- =¿ HelloServiceImpl generated interface
- =¿ HelloServiceImplService factory class

7. Client Applikation schreiben

Listing 19: Client

```
  package ch.fhnw.imvs.client;
  import ch.fhnw.imvs.client.jaxws.HelloServiceImpl;
3 import ch.fhnw.imvs.client.jaxws.HelloServiceImplService;

  public class Client {
    public static void main(String[] args) {
      HelloServiceImplService service =
8       new HelloServiceImplService();
      HelloServiceImpl port =
        service.getHelloServiceImplPort();
      String result = port.sayHello("Dominik");
      System.out.println(result);
13  }
  }
```

8. JAX-WS HTTP Request

Listing 20: ...

```
1 POST /hs HTTP/1.1           HTTP Request
  Accept: text/xml, multipart/related
  User-Agent: JAX-WS RI 2.2.4-b01
  Host: 127.0.0.1:9877
```

```
  Connection: keep-alive
6 Content-Length: 209
  Content-Type: text/xml; charset=utf-8   SOAP HTTP Binding
  SOAPAction: "http://server.jaxws.ds.fhnw.ch/HelloServiceImpl/sayHelloRequest"

  <?xml version="1.0" ?>          SOAP Payload
11 <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Body>
      <ns2:sayHello xmlns:ns2="http://server.jaxws.ds.fhnw.ch/">
        <name>Dominik</name>
      </ns2:sayHello>
16  </S:Body>
  </S:Envelope>
```

9. JAX-WS HTTP Response

Listing 21: ...

```
1 HTTP/1.1 200 OK           HTTP Response
  Transfer-encoding: chunked
  Content-type: text/xml; charset=utf-8
  Date: Mon, 18 Mar 2013 00:06:44 GMT
  Content-Length: 277
6
  <?xml version="1.0" ?>          SOAP Payload
  <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
  <ns2:sayHelloResponse
11 xmlns:ns2="http://server.jaxws.ds.fhnw.ch/">
  <return>
  Hello Dominik from SOAP at Mon Mar 18 01:06:44 CET 2013
  </return>
  </ns2:sayHelloResponse>
16 </S:Body>
  </S:Envelope>
```

3.3 Vergleich SOAP und XML-RPC

Feature	XML-RPC	SOAP
Structs	yes	yes
Arrays	yes	yes
Named structs & arrays	no	yes
Short learning curve	yes	no
Developer specified character set	no	yes
Developer defined data types	no	yes
Can specify recipient	no	yes
Require client understanding	no	yes
Message specific processing instructions	no	yes

4 Representation State Transfer (REST)

4.1 Prinzipien

- Addressability - Give everything an ID
- Uniform, Constrained Interface
- Representation-oriented
- Link things together - Use Resource references
- Stateless communications - Resources hold state
- Use standard HTTP methods

4.1.1 Addressability

- Resources = key abstractions in REST
- Each resource is addressable via a URI

Listing 22: REST example

```
http://www.example.com/customers/1234 http://www.example.com/customers?lastName=Meier http
    ://www.example.com/orders/2011/03/445245 http://www.example.com/products/ http://www.
    example.com/products/4711
```

4.1.2 CRUD

READ

- HTML: **GET, HEAD**
- Retrieve information in a particular representation
- No side effects, possibly cached
- May contain query parameters

CREATE

- HTML: **POST**
- Create a new sub-resource (without known ID)

CREATE & UPDATE

- HTML: **PUT**
- Update an existing resource
- Create a new resource with a known ID

DELETE

- HTML: **DELETE**
- Remove resources

OPTIONS

Returns allowed operations

4.1.3 Representation-oriented

Allow multiple representations of a resource:

- text/html
- text/plain
- application/json
- application/xml

4.1.4 Link things together

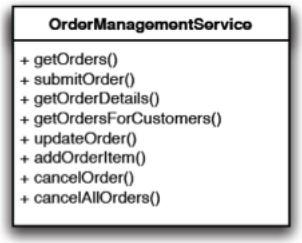
References to other resources may be used in representations

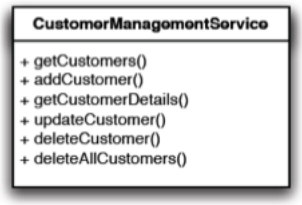
Listing 23: example

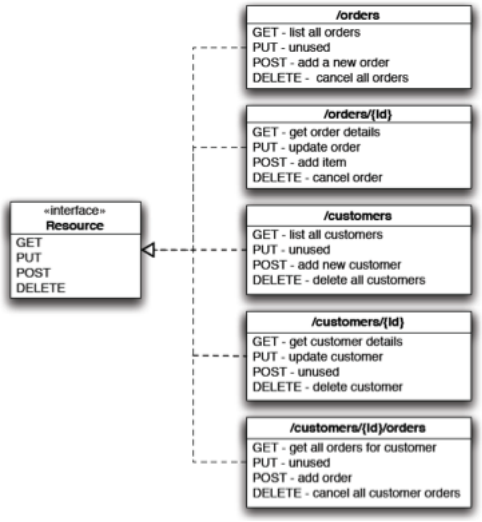
```
<order>
   <date>16.03.2013</date>
   <amount>23</amount>
   <product ref="http://example.com/products/4711" />
   <customer ref="http://example.com/customers/1234" />
</order>
```

4.2 SOAP vs REST

REST	SOAP
Resource Oriented	Service oriented
Messages represented in different formats	Messages represented in XML
HTTP used as protocol	Can be bound to different protocols
HTTP verbs are used for access and manipulation	Access and Manipulation is service specific
HTTP error codes are used as error messages	Fault Elements in SOAP body describes errors
No formal interface description language	WSDL is used as interface description language
GET requests can be cached in a proxy	All requests are POST requests, no caching!







4.3 JAX-RS

4.3.1 HTTP Methods

- Annotate resource class methods with HTTP method annotations @GET, @POST, @PUT, @DELETE, @HEAD
- Java method name has no significance
- Return value is mapped to the response (void = no response)

Listing 24: JAX-RS Service

```
@Singleton
@Path("/polls")
public class DoodlePoolResource {
  @GET
  @Path("{id}")
  public String getPoll(@PathParam("id") String key){ ... }

  @PUT
  @Path("{id}")
  public String putPoll(@PathParam("id") String key){ ... }

  @DELETE
  @Path("{id}")
  public void deletePoll(@PathParam("id") String key){ ... }
}
```

4.3.2 Injection

- Automatic Type Conversion from Strings to
  - Primitive types (int, short, float, double, byte, char, boolean)
  - Classes T which have a constructor with a single String parameter
  - Classes T which contain a static method T valueOf(String arg)
- Default values may be defined with @DefaultValue for the case that the parameter is not passed with the request

PathParam
Allows to extract values from URI template parameters

MatrixParam
Allows to extract matrix parameters (/images/cars;color=blue/2010/)

QueryParam
Allows to extract query parameters added to a URI

FormParam
Allows to extract values from posted form data

HeaderParam
Allows to extract request headers

CookieParam
Allows to extract values from HTTP cookies

4.3.3 Content Negotiation

@Produces declares type of result, default: all types are supported
@Produces({"text/plain", "text/html"})

@Consumes declares type which is accepted (PUT / POST)
@Consumes(”application/x-www-form-urlencoded”)

4.4 Data Binding

Listing 25: XStream Provider

```
@Provider
@Consumes("application/xstream")
@Produces("application/xstream")
public class XStreamProvider implements MessageBodyReader<Object>, MessageBodyWriter<Object
    >{
  private XStream xstream = new XStream(new DomDriver());

  public boolean isReadable(Class<?> type, Type genericType, Annotation[] annotations,
      MediaType mimeType) {
    return true;
  }
  public Object readFrom(Class<Object> type, Type genericType, Annotation[] annotations,
      MediaType mimeType, MultivaluedMap<String, String> httpHeaders, InputStream
      entityStream) {
    return xstream.fromXML(entityStream);
  }
  public boolean isWriteable(Class<?> type, Type genericType, Annotation[] ann, MediaType
      mimeType) {
    return true;
  }

  public long getSize(Object object, Class<?> type, Type genericType, Annotation[] ann,
      MediaType mimeType) {
    return -1;// size not yet known
  }

  public void writeTo(Object object, Class<?> type, Type genericType, Annotation[] ann,
      MediaType mimeType, MultivaluedMap<String, Object> httpHeaders, OutputStream
      entityStream) {
    xstream.toXML(object, entityStream);
  }
}
```

Listing 26: XStream Example

```
public class Client {
  public static void main(String[] args) {
  ClientConfig config = new DefaultClientConfig();
  config.getClasses().add(XStreamProvider.class);
  Client c = Client.create(config);
  WebResource r = c.resource("http://localhost:9998/msg");
  Msg msg = new Msg("Hello from XClient");
  r.type("application/xstream").put(msg);
  Msg res = r.accept("application/xstream").get(Msg.class);
  System.out.println(res);
  System.out.println(res.getText());
  System.out.println(res.getDate());
  }
}
```

4.5 JAXB

Java Architecture for XML Binding, kurz JAXB, ist eine Programmschnittstelle in Java, die es ermöglicht, Daten aus einer XML-Schema-Instanz heraus automatisch an Java-Klassen zu binden, und diese Java-Klassen aus einem XML-Schema heraus zu generieren. Diesen Vorgang nennt man XML-Datenbindung.

Listing 27: Unmarshalling

```
1
  JAXBContext jc = JAXBContext.newInstance("com.acme.foo:com.acme.bar");
  Unmarshaller u = jc.createUnmarshaller();
  FooObject fooObj = (FooObject) u.unmarshal(new File("foo.xml"));
  BarObject barObj = (BarObject) u.unmarshal(new File("bar.xml"));
```

Listing 28: Marshalling

```
  Marshaller m = jc.createMarshaller();
  m.marshal(fooObj, System.out);
```

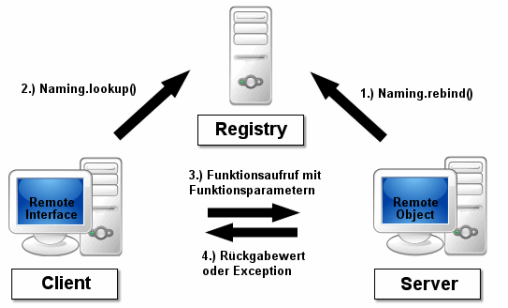
5 Remote Method Invocation

5.1 Einleitung

Remote Method Invocation (RMI) ist der Aufruf einer Methode eines entfernten Java-Objekts und realisiert die Java Art des Remote Procedure Call.
Auf der Client-Seite kümmert sich der sogenannte Stub um den Netzwerktransport. Der Stub muss entweder lokal oder über das Netz für den Client verfügbar sein. Das Erstellen des Stubs wird von der Java Virtual Machine übernommen.
Entfernte Objekte können zwar auch von einem bereits im Programm bekannten entfernten Objekt bereitgestellt werden, für die erste Verbindungsaufnahme werden aber die Adresse des Servers und ein Bezeichner (ein RMI-URL) benötigt. Für den Bezeichner liefert ein Namensdienst auf dem Server eine Referenz auf das entfernte Objekt zurück. Damit dies funktioniert, muss sich das entfernte Objekt im Server zuvor unter diesem Namen beim Namensdienst registriert haben. Der RMI-Namensdienst wird über statische Methoden der Klasse java.rmi.Naming angesprochen. Der Namensdienst ist als eigenständiges Programm implementiert und wird RMI Registry genannt.

5.2 Ablauf

Der Ablauf bei RMI:



1. Der Server registriert ein Remote Object bei der RMI-Registry unter einem eindeutigen Namen.
2. Der Client sieht bei der RMI-Registry unter diesem Namen nach und bekommt eine Objektreferenz, die seinem Remote Interface entsprechen muss.
3. Der Client ruft eine Methode aus der Objektreferenz auf. Dabei kann ein Objekt einer Klasse X übergeben werden, die der JVM des Servers bisher nicht bekannt ist. In diesem Fall lädt die Server-JVM die Klasse X dynamisch nach, beispielsweise vom Webserver des Client.
4. Die Server-JVM führt die Methode auf dem Remote Object aus, wobei evtl. dynamisch geladener Fremdcode benutzt wird, der im Allgemeinen Sicherheitsrestriktionen unterliegt. Dem Client werden die Rückgabewerte dieses Aufrufes gesendet, oder der Client bekommt eine Fehlermeldung.

5.3 Naming / Registry

bind (String, Remote)

- can only be called on localhost
- Binds a specified name to a remote object
- May throw AlreadyBoundException

rebind (String, Remote)

- can only be called on localhost
- Rebinds a specified name to a remote object

unbind (String)

- can only be called on localhost
- Destroys the binding for a specified name

lookup (String)

- Returns a reference to the remote object associated with the name

list (String)

- Lists the names present in a registry

5.4 Using RMI

5.4.1 Remote Interfaces

Define interfaces for remote classes:

- All remotable interfaces must extend interface java.rmi.Remote
- All methods must throw a java.rmi.RemoteException

Listing 29: Remote Interfaces

```
package ch.fhnw.ds.rmi.calculator;
public interface Calculator extends java.rmi.Remote {
  long add(long a, long b) throws java.rmi.RemoteException;
  long sub(long a, long b) throws java.rmi.RemoteException;
  long mul(long a, long b) throws java.rmi.RemoteException;
  long div(long a, long b) throws java.rmi.RemoteException;
}
```

5.4.2 Implementation

UnicastRemoteObject:

- Implementation extends class Unicast- RemoteObject or calls explicitly UnicastRemoteObject.exportObject
- Implementation must implement the defined remote interface

Listing 30: Implementation

```
package ch.fhnw.ds.rmi.calculator;
public class CalculatorImpl extends java.rmi.server.UnicastRemoteObject implements
    Calculator {
  public CalculatorImpl() throws java.rmi.RemoteException { }
  public long add(long a, long b) { return a + b; }
  public long sub(long a, long b) { return a - b; }
  public long mul(long a, long b) { return a * b; }
  public long div(long a, long b) { return a / b; }
}
```

5.4.3 stub and skeleton

Create stub and skeleton classes using the rmic compiler:

- No longer necessary in Java5 because Java5 adds support for the dynamic generation of stub classes at runtime
- rmic must still be used to pregenerate stub classes for remote objects that need to support clients running on earlier versions
- Dynamic Proxies
  - Implemented using java.lang.reflect.Proxy
  - Dynamic proxy is only used if no pregenerated stub class is available or if the system property java.rmi.server.ignoreStubClasses=true is set

– nly possible if clients run on Java 5 (or later)

5.4.4 Server

Create and compile the server application (registration):

- RMI service must be hosted in a server process
- RMI can use different naming services

Listing 31: Server

```
package ch.fhnw.ds.rmi.calculator;
import java.rmi.Naming;
public class Server {
public static void main(String args[]) throws Exception {
      Calculator c = new CalculatorImpl();
      Naming.rebind("CalculatorService", c);
   }
}
```

5.4.5 Client

Create and compile a client program to access the remote objects:

Listing 32: Client

```
package ch.fhnw.ds.rmi.calculator;
import java.rmi.Naming;
public class CalculatorClient {
  public static void main(String[] args) throws Exception {
        Calculator c = (Calculator)Naming.lookup("rmi://localhost/CalculatorService");
    System.out.println( c.sub(4, 3) );
    System.out.println( c.add(4, 5) );
    System.out.println( c.mul(3, 6) );
    System.out.println( c.div(9, 3) );
  }
}
```

5.4.6 Start of RMI-Registry & Server

Listing 33: Example Variant

```
package ch.fhnw.ds.rmi.calculator;
public class Server {
   public static void main(String args[]) throws Exception {
    try {
      LocateRegistry.createRegistry(1099);
        } catch (RemoteException e) {
      System.out.println(">> registry could not be exported");
      System.out.println(">> probably another registry already runs on 1099");
    }
        Calculator c = new CalculatorImpl(0x7777);
    Naming.rebind("rmi://localhost:1099/CalculatorService", c);
    System.out.println("Calculator server started...");
  }
}
```

Listing 34: List of all registered objects

```
public class ListRMIRegistry {
```

```
    public static void main(String[] args) throws Exception {
       if (args.length>0) {
          String host = args[0];
          if(args.length>1){
6       host = host + ":" + Integer.parseInt(args[1]);
        }
          String s[] = Naming.list("rmi://"+host);
          for(int i=0; i<s.length; i++) System.out.println(s[i]);
     } else {
11        System.out.println("Usage: java ListRMIREgistry " + "<host> [<port>]");
     }
   }
}
```

5.5 Marshalling

Transfer of parameters to remote object (input or output):

Parameters of primitive type (int, double, ...)
passed by value, in a machine-independent format

Parameters which are Serializable (e.g. String)
serializable objects are copied ⇒ call by value

Parameters which are Remote
only the reference to the remote object is passed, i.e. a new proxy is generated ⇒ call by reference

Parameters which are neither Serializable nor Remote
cannot be transferred (checked at runtime)

Listing 35: QuoteServer

```
1 public void addQuoteListener(QuoteListener c) {
    synchronized (clients) { clients.add(c); }
}
```

Listing 36: QuoteClient

```
  QuoteServer server = (QuoteServer)Naming.lookup("rmi://localhost/QuoteServer");
2 server.addQuoteListener(new AbstractQuoteListener() {
    public void update(String s) {
      System.out.println(s);
    }
});
```

5.6 Export / Unexport

UnicastRemoteObject:

Remote exportObject(Remote obj, int port)

- Exports remote object
- Makes it available to receive remote calls
- Returns the remote object stub

boolean unexportObject(Remote obj, boolean force)

- Removes remote object from the RMI runtime
- If force==false, object is only unexported if no calls are in progress or pending
- Returns true if operation is successful, false otherwise

Listing 37: Example Interface

```
public interface Counter extends Remote {
  public int getValue() throws RemoteException();
  public int increment() throws RemoteException;
  public int reset() throws RemoteException;
  // copies instance to a server
  public Counter migrateTo(String host) throws RemoteException;
  // copies counter back to the caller
  public Counter migrateBack() throws RemoteException;
}
```

Listing 38: Migrator interface

```
public interface Migrator extends Remote {
  public Counter migrate(Counter counter) throws RemoteException;
}
```

Listing 39: Migrator implementation

```
public class MigratorImpl extends UnicastRemoteObject implements Migrator {
  public MigratorImpl() throws RemoteException { }
  public Counter migrate(Counter counter) throws RemoteException {
    UnicastRemoteObject.exportObject(counter, 0);
    return counter; // Variant: return the proxy
  }
}
```

Listing 40: Counter implementation

```
public class CounterImpl implements Counter, Serializable {
  private int value;
  public int getValue() { return value; }
  public int increment() { value++; return value; }
  public int reset() { value = 0; return value; }
  public Counter migrateTo(String host) throws RemoteException {
    try {
      Migrator migrator = (Migrator) Naming.lookup("rmi://" + host + "/Migrator");
      return migrator.migrate(this);
    } catch(Exception e) { throw new RuntimeException(e); }
  }
  public Counter migrateBack() throws RemoteException {
    UnicastRemoteObject.unexportObject(this, true);
    return this;
  }
}
```

5.7 Threads

5.7.1 Specification

A method dispatched by the RMI runtime to a remote object implemen- tation may or may not execute in a separate thread. The RMI runtime makes no guarantees with respect to mapping remote object invocations to threads. Since remote method invocation on the same remote object may execute concurrently, a remote object implementation needs to make sure its implementation is thread-safe

5.7.2 Consequence

- You have to assume that your objects are called from different threads

- You have to implement objects thread-safe

5.8 Dynamically Loaded Classes

Required classes can be loaded over the network
E.g. provided by a webserver

Class Loading / Security

- A special class loader is provided: RMIClassLoader
- Security manager has to support remote class loading
  System.setSecurityManager(new RMISecurityManager())

Start of RMI-Registry

- rmiregistry must not contain the needed classes in its path!!
- Specify the java.rmi.server.useCodebaseOnly=false option (since Java7)
  rmiregistry -J-Djava.rmi.server.useCodebaseOnly=false

Start of Server

- Specify codebase for downloading class files
- java -Djava.rmi.server.codebase=http://localhost:8080 ch.fhnw.ds.rmi.calculator.Server

Start of Client

- permission to access server has to be provided (due to security manager)
  java -Djava.security.policy=policy.txt ch.fhnw.ds.rmi.calculator.Client localhost
- policy file
  grant { permission java.net.SocketPermission "10.223.240.137:1024-", "connect,resolve"; };

5.9 Socket Factories

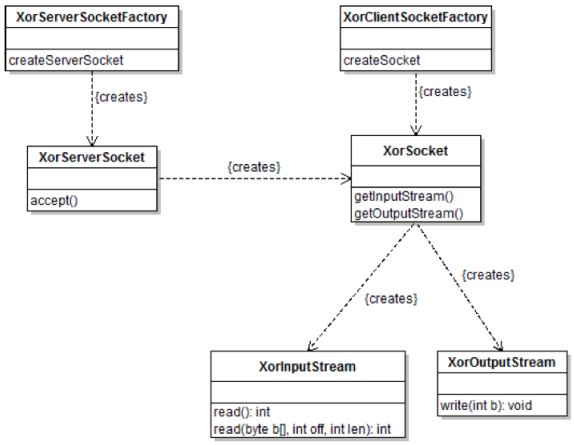
RMI Servers

- When RMI object is exported, then a ServerSocket is created
- Access to a RMI object is established over a Socket
- Several objects may be exported over the same ServerSocket

Socket Factories

- Provide your own ServerSockets and Sockets to RMI
- Applications
  Authentication, Encryption, Compression, HTTP tunneling, time-out control
- Factories
  ServerSocketFactory (returns ServerSocket), ClientSocketFacotory (returns Socket, is downloaded to client)
- Specified with exportObject

5.10 Example: XOR encoding



Listing 41: XorServerSocketFactory

```
public class XorServerSocketFactory implements RMIServerSocketFactory {
  private final byte pattern;
  public XorServerSocketFactory(byte pat) {this.pattern = pat;}
  public ServerSocket createServerSocket(int port) throws IOException {
    return new XorServerSocket(port, pattern);
  }
  public int hashCode() { return (int) pattern; }
  public boolean equals(Object obj) {
    return obj != null && getClass() == obj.getClass() && pattern == ((
        XorServerSocketFactory)obj).pattern);
  }
}
```

Listing 42: XorClientSocketFactory

```
public class XorClientSocketFactory implements RMIClientSocketFactory, Serializable {
  private final byte pattern;
  public XorClientSocketFactory(byte pat) {this.pattern = pat;}
  public Socket createSocket(String host, int port) throws IOException {
    return new XorSocket(host, port, pattern);
  }
  public int hashCode() {return (int) pattern;}
  public boolean equals(Object obj) {
  return obj != null && getClass() == obj.getClass() && pattern == ((XorClientSocketFactory
      ) obj).pattern;
  }
}
```

Listing 43: XorServerSocket

```
class XorServerSocket extends ServerSocket { private final byte pattern;
  public XorServerSocket(int port, byte pattern) throws IOException {
    super(port); this.pattern = pattern;
  }
  public Socket accept() throws IOException {
    Socket s = new XorSocket(pattern); implAccept(s); return s;
  }
}
```

Listing 44: XorSocket

```
class XorSocket extends Socket {
  private final byte pat;
  private InputStream in = null;
```

```
    private OutputStream out = null;
    public XorSocket(byte pat) throws IOException {
      super(); this.pat = pat;
    }
    public XorSocket(String host, int port, byte pat) throws IOException {
      super(host, port); this.pat = pat;
    }
    public synchronized InputStream getInputStream() throws IOException {
      if (in == null)
        in  = new XorInputStream(super.getInputStream(), pat);
      return in;
    }
    public synchronized OutputStream getOutputStream() throws IOException {
          if (out == null)
            out = new XorOutputStream(super.getOutputStream(),pat);
      return out;
    }
  }
```

Listing 45: XorOutputStream

```
  class XorOutputStream extends FilterOutputStream { private final byte pattern;
    public XorOutputStream(OutputStream out, byte pattern) {
      super(out); this.pattern = pattern;
    }
    public void write(int b) throws IOException {
      out.write((b ^ pattern) & 0xFF);
    }
  }
```

Listing 46: XorOutputStream

```
  class XorInputStream extends FilterInputStream {
    private final byte pattern;
    public XorInputStream(InputStream in, byte pattern) {
      super(in); this.pattern = pattern;
    }
    public int read() throws IOException {
      int b = in.read();
      if (b != -1) // if not EOF or an error
            b = (b ^ pattern) & 0xFF;
          return b;
    }
    public int read(byte b[], int off, int len) throws IOException {
      int numBytes = in.read(b, off, len);
      for (int i = 0; i < numBytes; i++) {
        b[off + i] = (byte) ((b[off + i] ^ pattern) & 0xFF);
      return numBytes;
    }
  }
```

Listing 47: Use XorFactories

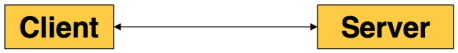
```
  public class CalculatorImpl extends java.rmi.server.UnicastRemoteObject implements rmi.
      calculator.Calculator {
    public CalculatorImpl(int port) throws java.rmi.RemoteException{
      this(port, (byte)0xAC);
      }
    public CalculatorImpl(int port, byte pat) throws java.rmi.RemoteException {
      super(port, new XorClientSocketFactory(pat), new XorServerSocketFactory(pat));
    }
    public long add(long a, long b) { return a + b; }
    public long sub(long a, long b) { return a - b; }
    public long mul(long a, long b) { return a * b; }
    public long div(long a, long b) { return a / b; }
  }
```

6 Asynchronous Communication

6.1 Prinzip

6.1.1 Synchronous Communication Model

Client and Server are directly coupled (Telephone Connection)



6.1.2 Asynchronous Communication Model

Client and Server communicate over a 3rd party Message Oriented Middleware (MOM). (SMS, Email)

6.2 Advantages

- Loosely coupled communication
- Sender/receiver do not have to be active at the same time and may be disconnected
- Sender/receiver only have to agree on message format, no dependency on interfaces

6.3 Model

- Producer sends a message to a virtual channel
- Producer does not have to wait for a reply
- Producer does not know who will receive the message
- Producer is not dependent on the availability of consumers
- Transaction and security context of a producer are not propagated to the consumer of a message

7 Java Message Service

7.1 Point-to-Point Messaging Domain

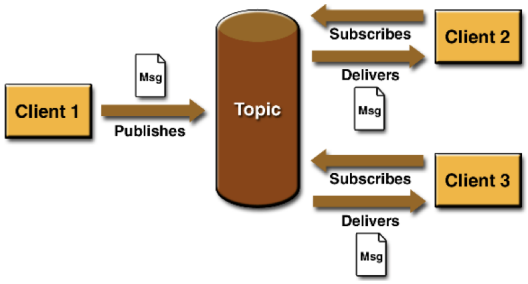
Queue:

- Every message is sent to a particular queue
- Consumer fetches message at a particular queue
- Message is kept until it is consumed/expired

Characteristics:

- Each message is consumed by only one client
- No time dependency between sender and consumer
- Consumer acknowledges consumption of a message
- Queues are persistent

7.2 Publish/Subscribe Messaging Domain

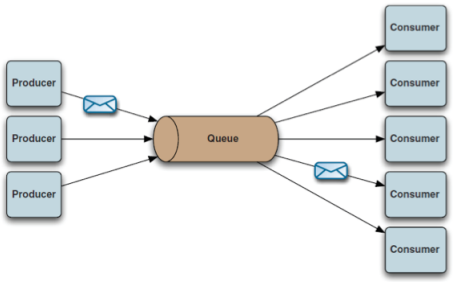


Characteristics:

- Every message can be consumed by several clients
- Time dependency, only those messages are consumed which arrived after registration
- Consumer has to stay active
- A durable connection allows con- sumers to disconnect and later re- connect and collect messages that were published meanwhile

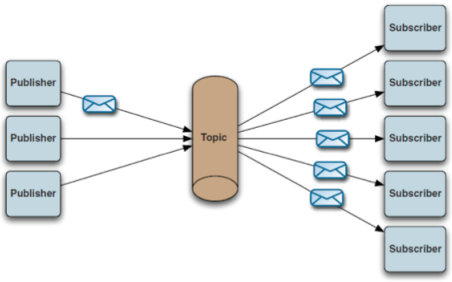
7.3 Destinations

Queue



One-to-one message paradigm

Topic



One-to-many message paradigm

7.4 Message Consumption

Synchronously

JMS Client can receive messages explicitly

blocking mode receive()

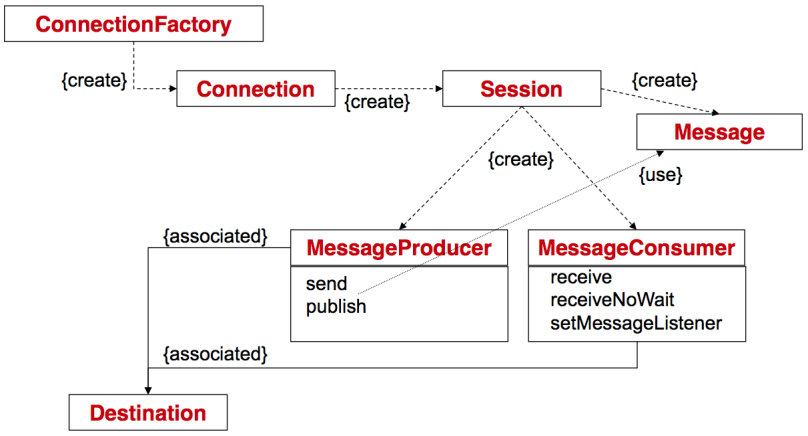
time-out mode receive(int timeout)

Asynchronously

JMS Client can register a message listener which is invoked when a message arrives
setMessageListener()

7.5 JMS API

7.5.1 Overview



7.5.2 Destinations

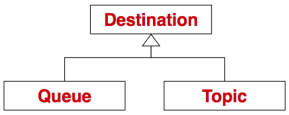
Destinations:

- Destination where messages are delivered to or received from

  Queue point-to-point messaging domain destination

  Topic publish/subscribe messaging domain destination

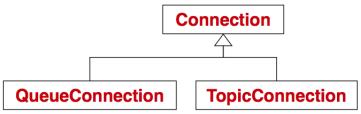
- A JMS application can work with several destinations
- New destinations are created using an administration tool
- Destinations are accessed using JNDI



7.5.3 Connections

Connections:

- Represents a communication channel to the JMS provider
- Connection has to be started before use (before receiving messages)
- Connection has to be closed when the program is done using it



Connection Factories:

- Connections are created from connection factories
- Factories are accessed via JNDI

- Java EE: two factories preconfigured

Connection.start:

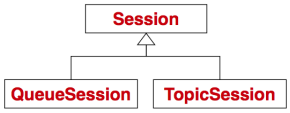
- Until connection is started, message flow does not occur
- Connection has to be started before messages can be transmitted or received

Connection.close:

- Clients should close a connection as a provider typically allocates significant resources outside the JVM on behalf of a connection
- There is no need to close the sessions, producers, and consumers of a closed connection
- Closing a connection causes all temporary destinations to be deleted

7.5.4 Sessions

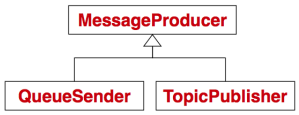
Sessions:



- Single-threaded context to deliver and consume message
  Any message sending and receiving happens in a serial order, one-by-one
- Transactional
  A set of send and receive operations can be packed in an atomic operation
- Sessions are created from connections
  createQueueSession(boolean transacted, int acknowledgeMode)
  createTopicSession(boolean transacted, int acknowledgeMode)

7.5.5 Message Producers

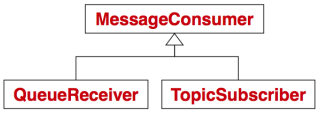
Message Producers:



- Used to send or publish message
- TCreated from session object (createSender / createPublisher)
- Producer is associated with a destination (queue or topic)
  queueSession.createSender(queue)
  topicSession.createPublisher(topic)
- Delivery of message with send / publish

7.5.6 Message Consumers

Message Consumers:



- Used to receive messages
- Created from session object (createReceiver / createSubscriber)
- Consumer is associated with a destination (queue or topic)
- Receipt of message with receive()/receiveNoWait()
- Deactivation with close()
- Messages are received after the connection is started
- Message filter can be specified for message consumers

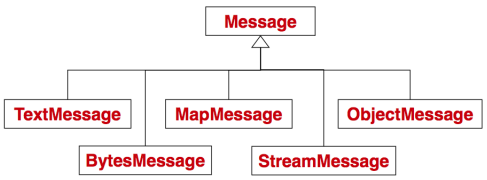
7.5.7 Message Listeners

- comparable to event listener (interface MessageListener { void onMessage(Message msg); })
- Can be registered both in a QueueReceiver and a TopicSubscriber
- Per session only one listener is active

7.5.8 Messages

Message Parts:

- Header
- Properties [optional]
- Body [optional]



Header Fields:

JMSDestination	send/publish		
JSMDeliveryMode	send/publish	int	persistency
JMSExpiration	send/publish	long	time() + TTL msec
JMSPriority	send/publish	int	0 (low) .. 9 (high)
JMSMessageID	send/publish		
JMSTimestamp	send/publish		
JMSCorrelationID	client		
JMSReplyTo	client		
JMSType	client	String	
JMSRedelivered	JMS provider		

Properties:

- additional properties can be specified by client (String/primitive - pairs)

Body depends on message type:

TextMessage String (getText/setText)

MapMessage set of name/value pairs (String/primitive)

BytesMessage byte stream

StreamMessage sequence of primitive data types

ObjectMessage contains one serializable object

messages are created over a session object, there exist 5 factory methods:

- TextMessage m = session.createTextMessage();
- MapMessage m = session.createMapMessage();
- BytesMessage m = session.createBytesMssage();
- StreamMessage m = session.createStreamMessage();
- ObjectMessage m = session.createObjectMessage();

7.6 Sending / Receiving a Message

1. Obtain a connection factory that provides JMS connections to the messaging system
2. Obtain a topic or a queue that represents a destination address for the message
3. Create an connection to the JMS provider using the factory

4. Create a session that is used to group the actions of sending and receiving messages (single-threaded context)
5. Create a publisher or a sender that sends messages to the specified topic or queue
6. create a subscriber or a receiver and optionally register a MessageListener that receives messages from the specified topic or queue

7.7 Examples

7.7.1 Simple Example

Listing 48: Simple Example Sender

```
public class SimpleQueueSender {
  public static void main(String[] args) throws Exception {
    Context jndiContext = new InitialContext();
    QueueConnectionFactory queueConnectionFactory = (QueueConnectionFactory) jndiContext.
        lookup("QueueConnectionFactory");
    Queue queue = (Queue) jndiContext.lookup(args[0]);
    QueueConnection qconn = null;
    try {
      qconn = queueConnectionFactory.createQueueConnection();
      QueueSession queueSession = qconn.createQueueSession(false, Session.AUTO_ACKNOWLEDGE)
          ;
      QueueSender queueSender=queueSession.createSender(queue);
      TextMessage m = queueSession.createTextMessage();
      for (int i = 0; i < 3; i++) {
        m.setText("This is message " + (i + 1));
        queueSender.send(m);
      }
    } finally {
      if (qconn != null)
        try { qconn.close(); } catch(JMSException e){}
    }
  }
}
```

Listing 49: Simple Example Receiver

```
public class SimpleQueueReceiver {
  public static void main(String[] args)  throws Exception {
    Context jndiContext = new InitialContext();
    QueueConnectionFactory queueConnectionFactory = (QueueConnectionFactory) jndiContext.
        lookup("QueueConnectionFactory");
    Queue queue = (Queue) jndiContext.lookup(args[0]);
    QueueConnection qConn = null;
    try {
      qConn = queueConnectionFactory.createQueueConnection();
      QueueSession queueSession = qConn.createQueueSession(false, Session.AUTO_ACKNOWLEDGE)
          ;
      QueueReceiver queueReceiver = queueSession.createReceiver(queue);
      qConn.start();
      while (true) {
        Message m = queueReceiver.receive();
        if (m instanceof TextMessage) {
          TextMessage message = (TextMessage) m;
          System.out.println("Reading message: " + message.getText());
        }
      }
    } finally {
      if (qConn != null)
        try {qConn.close();} catch (JMSException e) {}
    }
  }
}
```

7.7.2 Echo

Listing 50: Echo Client

```
public static void main(String[] args) throws Exception{
  Hashtable<String,String> properties = new Hashtable<String,String>();
  ...
  Context context = new InitialContext(properties);
  QueueConnectionFactory factory = (QueueConnectionFactory) context.lookup("
      ConnectionFactory");
  Queue queue = (Queue) context.lookup("ECHO");
  QueueConnection connection=factory.createQueueConnection("","");
  QueueSession session = connection.createQueueSession(false, QueueSession.AUTO_ACKNOWLEDGE
      );
  QueueSender sender  = session.createSender(queue);
  TemporaryQueue tempQueue = session.createTemporaryQueue();
  QueueReceiver receiver = session.createReceiver(tempQueue);
  connection.start();
  TextMessage request = session.createTextMessage();
  request.setText("Hello World [" + new Date() + "]");
  request.setJMSReplyTo(tempQueue); sender.send(request);
  TextMessage response = (TextMessage) receiver.receive();
  System.out.println("Response: " + response.getText());
  connection.close();
}
```

Listing 51: Echo Server

```
public static void main(String[] args) throws Exception{
  Hashtable<String,String> properties = new Hashtable<String,String>();
   ...
  Context context = new InitialContext(properties);
  QueueConnectionFactory factory = (QueueConnectionFactory) context.lookup("
      ConnectionFactory");
  Queue queue = (Queue) context.lookup("ECHO");
  QueueConnection connection=factory.createQueueConnection("","");
  QueueSession session = connection.createQueueSession(false, QueueSession.AUTO_ACKNOWLEDGE
      );
  QueueReceiver receiver = session.createReceiver(queue);
  connection.start();
  System.out.println("Echo service is running...");
  while(true){
    TextMessage request = (TextMessage)receiver.receive();
    Queue replyQueue = (Queue) request.getJMSReplyTo();
    TextMessage response = session.createTextMessage();
    response.setText("Echo: "+request.getText());
    QueueSender sender = session.createSender(replyQueue);
    sender.send(response);
    System.out.println("Handled "+request.getText());
  }
}
```

7.8 Guaranteed Messaging and Transactions

7.8.1 Delivery Mode

PERSISTENT (Default)
: Instructs the JMS provider to take extra care to ensure that message is not lost in case of a JMS provider failure

NON_PERSISTENT
: Does not require the JMS provider to store the message

7.8.2 Specification of delivery mode:

On MessageProducer
producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT)

On send or publish method
producer.send(msg, DeliveryMode.NON_PERSISTENT, 4, 10000)

- Priority level: 0 (lowest), 4 (default), 9 (highest)
- Message expiration in msec

7.8.3 Durable Subscribers

Nondurable subscribers
Receive messages only when they are actively listening on a topic

Durable subscribers

- Receive all messages sent to that topic (store-and-forward messaging)
- Messages are stored under a subscriber ID (ClientID-SubscriptionName)
- Durable subscribers must perform two initial steps:
  - Set the ClientID on the connection (before creating any sessions).
    The ClientID is part of the durable subscriber's name (a client can only be connected once)
  - Use the createDurableSubscriber factory method of the topic session. This method requires a second parameter with the name of the durable subscriber
- Durable subscribers may be unsubscribed

7.8.4 Message Acknowledgements

- Servers acknowledge the receipt of messages from producers
- Consumers acknowledge the receipt of messages from servers

Modes:

conn.createQueueSession(false, Session.CLIENT_ACKNOWLEDGE);

conn.createTopicSession(false, Session.CLIENT_ACKNOWLEDGE);

AUTO_ACKNOWLEDGE
Session automatically acknowledges receipt of a message

DUPS_OK_ACKNOWLEDGE
Same as AUTO, but messages may be delivered more than once

CLIENT_ACKNOWLEDGE

- Client acknowledges message by calling the message's acknowledge method
- Acknowledging a message acknowledges the receipt of ALL messages of that session

Duplicate Messages:

- With AUTO / DUPS_OK acknowledgement modes acknowledgement may get lost
- To guard against duplicate messages an application must check whether a redelivered message was already processed
- JMSMessageId header is unique and can be used as a key in a table

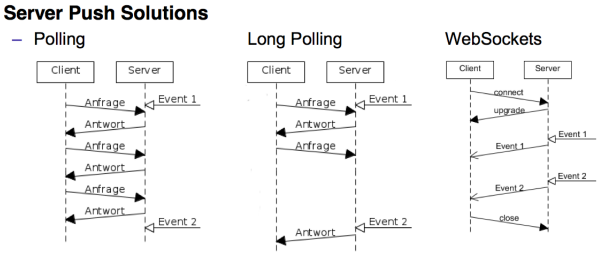
7.8.5 JMS Transactions

- conn.createQueueSession(true, Session. SESSION_TRANSACTED);
- conn.createTopicSession(true, Session. SESSION_TRANSACTED);
- All messages sent or received using such a session are automatically grouped in a transaction
- The transactioni remains open until either a session.rollback() or a session.commit() is invoked (which also starts a new transaction)
- Transactions guarantee
  - That message is delivered from the client to the JMS provider
  - That message is delivered from the JMS provider to the consumer

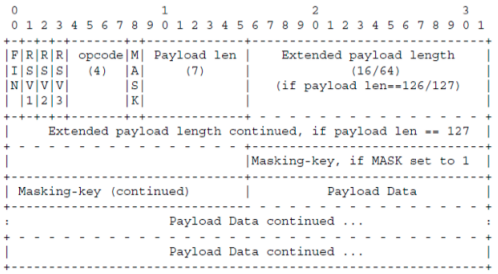
8 WebSockets

8.1 Spec

8.1.1 Bidirectional Communication



8.1.2 Format



8.1.3 Handshake

```
GET /examples/websocket/echoStream HTTP/1.1 Host: server.example.com
Connection: Upgrade
Upgrade: websocket
Sec-Websocket-Key: mqn5Pm7wtXEX6BzqDInLjw==
Sec-Websocket-Version: 13

HTTP/1.1 101 Switching Protocols
Server: Apache-Coyote/1.1
Upgrade: websocket
Connection: upgrade
Sec-WebSocket-Accept: +TdGPOkAq62+toDOhVGj2QZWwg8= Date: Thu, 04 Apr 2013 19:21:39 GMT
```

8.1.4 Negotiation

```
GET /examples/websocket/echoStream HTTP/1.1 Host: server.example.com
Connection: Upgrade
Upgrade: websocket
Sec-Websocket-Key: mqn5Pm7wtXEX6BzqDInLjw==
Sec-WebSocket-Protocol: chat, superchat
Sec-Websocket-Version: 13

HTTP/1.1 101 Switching Protocols
Server: Apache-Coyote/1.1
Upgrade: websocket
Connection: upgrade
Sec-WebSocket-Accept: +TdGPOkAq62+toDOhVGj2QZWwg8= Sec-WebSocket-Protocol: chat
Date: Thu, 04 Apr 2013 19:21:39 GMT
```

8.2 Java-WebSocket

Listing 52: Echo Server

```
public class EchoServer extends WebSocketServer {
  public static void main(String[] args) {
    WebSocketServer server = new EchoServer(80);
    server.start();
  }
  public EchoServer( int port ) {
    super( new InetSocketAddress(port) );
  }
  public void onOpen(WebSocket c, ClientHandshake handshake) {
    System.out.println("onOpen " + c.getRemoteSocketAddress());
  }
  public void onClose(WebSocket c, int id, String s, boolean rem){
        System.out.println("closed " + c.getRemoteSocketAddress());
    }
  public void onMessage(WebSocket c, String message) {
    System.out.println("onMessage from" + c.getRemoteSocketAddress() + ": " + message );
    c.send("Echo: " + message);
  }
  public void onError(WebSocket c, Exception ex) {
    System.out.println("onError " + c + ":" + ex );
  }
}
```

Listing 53: Echo Client

```
public class EchoClient {
  public static void main(String[] args) throws Exception {
    final URI url = new URI("ws://echo.websocket.org/");
    final WebSocketClient c = new WebSocketClient(url, new Draft_17()) {
      public void onOpen(ServerHandshake handshakedata) {
        System.out.println("onOpen");
        send("Hello");
      }
      public void onMessage(String message) {
              System.out.println("onMessage "+message);
        close();
      }
      public void onError(Exception ex) {
        System.out.println("onError");
      }
      public void onClose(int code, String txt, boolean remote) {
        System.out.println("onClose");
      }
    };
        c.connect();
  }
}
```

8.2.1 Tomcat

Listing 54: Echo Server

```
public abstract class WebSocketServlet() {
  protected void doGet(HttpServletRequest req, HttpServletResponse resp) {...}
  protected abstract StreamInbound createWebSocketInbound( String subProtocol,
      HttpServletRequest request);
  protected String selectSubProtocol(List<String> subProtocols)
  protected boolean verifyOrigin(String origin)
}
public abstract class MessageInbound extends StreamInbound {
  protected abstract void onBinaryMessage(ByteBuffer message);
  protected abstract void onTextMessage(CharBuffer message);
  protected void onOpen(WsOutbound outbound) {}
  protected void onClose(int status) {}
```

```
    protected void onPong(ByteBuffer payload) {}
13 }
  public class EchoMessage extends WebSocketServlet {
    protected StreamInbound createWebSocketInbound(
      String subProtocol, HttpServletRequest request) {
        return new EchoMessageInbound();
18    }
      private class EchoMessageInbound extends MessageInbound {
      protected void onBinaryMessage(ByteBuffer message) throws IOException {
        getWsOutbound().writeBinaryMessage(message);
          }
23        protected void onTextMessage(CharBuffer message) throws IOException {
            getWsOutbound().writeTextMessage(message);
      }
    }
  }
```

9 Tuple Spaces

9.1 Prinzip

9.1.1 Distributed Shared Associative Memory Space

- Provides a repository of tuples that can be accessed concurrently
  Associative: items are selected by reference to their contents
- Implements blackboard metaphor
  - Producers post their data as tuples in the space
  - Consumers retrieve data from the space that match a certain pattern

9.1.2 Uncoupling

Time: Tuples have their own life span (independent of producers and consumers)

Destination: The producer requires no knowledge about the future of a tuple, i.e. tuple space communication is fully anonymous

Space: Tuple space provides a globally shared data space to all processes

9.2 Overview

- Java implementation of the Linda coordination model
- Enables communication between applications in a network of heterogeneous computers and operating systems
- Features:
  - Blocking and non-blocking tuple set operators
  - Persistent data repository
  - Database indexing and query capabilities
  - Event notifications
  - Access controls

9.3 Basic Definitions

9.3.1 Tuple

- A tuple is an ordered sequence of fields
- Template tuple is a tuple used for matching

9.3.2 Field

- Type
- Value
- Name (optional)
  If a name is specified, then this field is indexed

9.3.3 TupleSpace

- Implements a space where tuples can be added/removed
- It is stored and administered across a network on one or more "TSpace Servers"(a TSpace server may contain many TupleSpaces)
- Several threads on the same or different machines can be accessing the space simultaneously
- For each different TupleSpace a client wishes to access, it requires a separate instance of this class, even if they are managed by the same server
- In order to obtain an instance of a specific tuple space you need the name of the space, and the name of a server that manages that space (and a port)

Listing 55: TupleSace

```
TupleSpace(String name, String host);
TupleSpace(String name, String host, int port);
```

9.4 Primitive operations

write(Tuple tuple)
Adds a tuple to the store

take(Tuple template) non-blocking

- Searches for a tuple that matches the template
- When found, the tuple is removed from the space and returned
- If none is found, returns null

waitToTake(Tuple template) blocking
Like take, but blocks until a match is found

read(Tuple template) non-blocking
Like take, except that the tuple is not removed from the space

waitToRead(Tuple template) blocking
Like waitToTake, except that the tuple is not removed from the space

scan(Tuple template) non-blocking

- Like read, except that it returns all tuples that match
- The tuples read are returned in one tuple (with the tuples as fields)

countN(Tuple template) non-blocking

- Like scan, except that it returns the number of matching tuples only
- Not that useful as the number of tuples may have changed after invocation

9.5 Tuple

9.5.1 Ways to create a tuple

Listing 56: Ways to create a tuple

```
Tuple t = new Tuple("key", "value");
Tuple t = new Tuple(); t.add("key"); t.add("value");
Tuple t = new Tuple(new Field("key"), new Field("value"));
Tuple t = new Tuple(new Field(String.class, "key"), "value");
```

```
Tuple t = new Tuple(new Field("key", String.class, "key"),new Field("value", String.class,
    "value"));
ts.write("key", "value");
```

9.5.2 Ways to read a tuple

- In order to read a tuple you must create a template Tuple
- A template Tuple is a Tuple that is used to select matching Tuples
- It will contain 0 or more fields that are either actual fields or formal fields
- Actual fields
  have a value that must be matched exactly against the corresponding Fields in the tuples that are in the space
- Formal fields
  - have a class but no value and only describe the type of value that is to be returned
  - Basically they act as wildcards

Listing 57: Ways to read a tuple

```
Tuple template = new Tuple("key", "value"); Tuple t = ts.read(template);
Tuple template = new Tuple("key", new Field(String.class));
Tuple t = ts.read(template);
Tuple template = new Tuple(new Field(String.class),
                           new Field(String.class));
Tuple t = ts.read(template);
Tuple t = ts.read(new Field(String.class),
                  new Field(String.class));
```

9.5.3 Echo Example

Listing 58: Echo Client

```
public class Client {
  public static void main(String[] args) throws Exception {
    TupleSpace ts = new TupleSpace("Echo", "localhost");
    UUID uuid = UUID.randomUUID();
    TupleID id = ts.write(new Tuple(uuid, "Hello World"));
    Tuple res = ts.waitToTake(new Tuple(new Field(String.class), uuid, new Field(String.
        class)));
    System.out.println(res.getField(2).getValue());
  }
}
```

Listing 59: Echo Server

```
public class Server {
  public static void main(String[] args) throws Exception {
    TupleSpace ts = new TupleSpace("Echo", "localhost");
    while (true) {
          Tuple template = ts.waitToTake(new Field(UUID.class), new Field(String.class));
      String text = (String) template.getField(1).getValue();
      ts.write(new Tuple("Response", template.getField(0), "Echo: " + text));
    }
  }
}
```

9.5.4 Factorial Example

Listing 60: Master

```
public class Master {
  public static void main(String[] args) throws Exception {
    TupleSpace ts = new TupleSpace("Factorial", "localhost");
    ts.deleteAll();
    int n = 100;
    ts.write("counter", 1);
    for (int i = 0; i < n; i++) {
      ts.write("x", BigInteger.valueOf(i + 1));
      if (i > 0) ts.write("task");
    }
    Tuple res = ts.waitToTake(new Tuple("counter", n));
    res = ts.take(new Tuple("x", new Field(BigInteger.class)));
    System.out.println(n+"! = " + res.getField(1).getValue());
  }
}
```

Listing 61: Worker

```
public class Worker {
  public static void main(String[] args) throws Exception {
    TupleSpace ts = new TupleSpace("Factorial", "localhost"); while (true) {
      ts.waitToTake("task");
      Tuple a1=ts.waitToTake("x", new Field(BigInteger.class));
      Tuple a2=ts.waitToTake("x", new Field(BigInteger.class));
      BigInteger x1 = (BigInteger) (a1.getField(1).getValue());
      BigInteger x2 = (BigInteger) (a2.getField(1).getValue());
      ts.write("x", x1.multiply(x2));
      Tuple c = ts.waitToTake("counter",new Field(Integer.class));
      ts.write("counter",(Integer) c.getField(1).getValue()+1);
    }
  }
}
```

9.6 Transactions

- Create an instance of Transaction class
- Call method beginTrans on the transaction
- Add a TupleSpace object to the transaction object using addSpace
- Perform operations (write, read, ..) on the TupleSpace object added to the transaction
- Call commitTrans on the transaction to make the effects of operations persistent
- Call abortTrans method of Transaction object to undo all operations that belong to the current transaction
- A transaction may be added to several tuple spaces!

Listing 62: Transaction Example

```
Transaction trans = new Transaction();
TupleSpace ts = new TupleSpace("testing");
trans.addSpace(ts);
trans.beginTrans(); // start of transaction
ts.write("email", "to", "stefan.hoechli@fhnw.ch");
ts.write("email", "text", "Hello");
ts.write("email", "from", "dominik.gruntz@fhnw.ch");
trans.commitTrans(); // end of transaction
```

9.7 Event Registration

9.7.1 Event notifications

- A mechanism that enables clients to register with the server in order to be informed when specific types of events are available
  - WRITE
  - DELETE
- Event registration can be cancelled with eventDeRegister

Listing 63: Event notifications Example

```
TupleSpace ts = new TupleSpace("testing");
Tuple template = new Tuple(new Field(String.class));
boolean newThread = false;
int id = ts.eventRegister("WRITE", template, callback, newThread);
...
ts.eventDeRegister(id);
```

Listing 64: Callback interface

```
interface Callback {
  public boolean call(
    String commandName,   // e.g. WRITE or DELETE
    String tsName,        // the name of the tuple space this
                          // command was executed on.
    int sequenceNumber,   // the registration id for the
                          // executed command
    SuperTuple tuple,     // the returned tuple or a tuple
                          // which contains an exception boolean
    isException           // indicates whether the command
                          // was processed normally or with
                          // an exception
  );
}
```

10 Remoting Patterns

10.1 Remoting Styles

10.1.1 Remote Procedure Calls

Synchronous call
Client waits until invocation succeeds (asynchronous variants exist)

Remoting errors
In contrast to in-process operations remoting errors may occur

Procedure-RPC vs. OO-RPC

Procedural:

- Server provides a set of operations
- Typically stateless services

OO-RPC:

- Server hosts a set of objects ⇒ object identity
- Distributed object typically have their own state

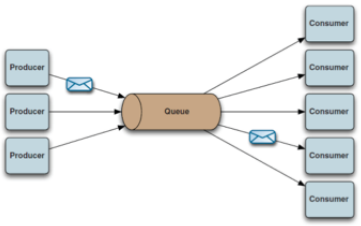
10.1.2 Message Passing

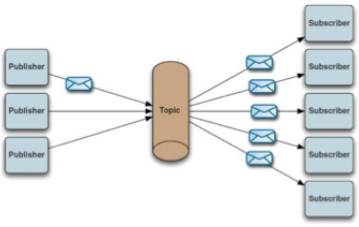
Message Queue Service

- Sender and Receiver decoupled by message queue servic
- Sender and receiver do not need to know each other
- Reliability
  - Exactly-once-semantics for queues
  - Acknowledgement of message consumption

Message Styles

- Queue ⇒ one receiver
- Topic ⇒ many receivers





10.1.3 Shared Repository

- Provides a small interface consisting of access primitives to the clients (CRUD)
- Idea based on (Linda's) Tuple Spaces

Samples: TupleSpaces, DataBases, REST

10.2 Pattern Language for RPC-like communication

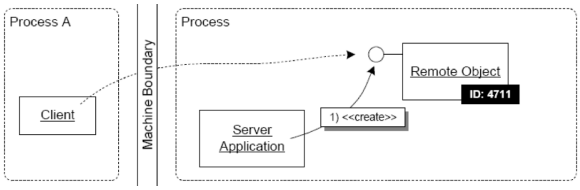
10.2.1 Pattern Language

- Collection of design patterns for a particular application domain
- Describes how to build a complex system of a certain type – too complicated to be described in one pattern
- Patterns describe roles, need not necessarily form its own component, a component may play multiple roles at the same time

10.2.2 Remote Object Access

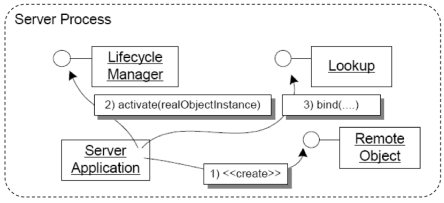
Remote Object:

- Is accessed from client (over a well-defined interface) across machine boundary
- Has a unique object ID in its local address space



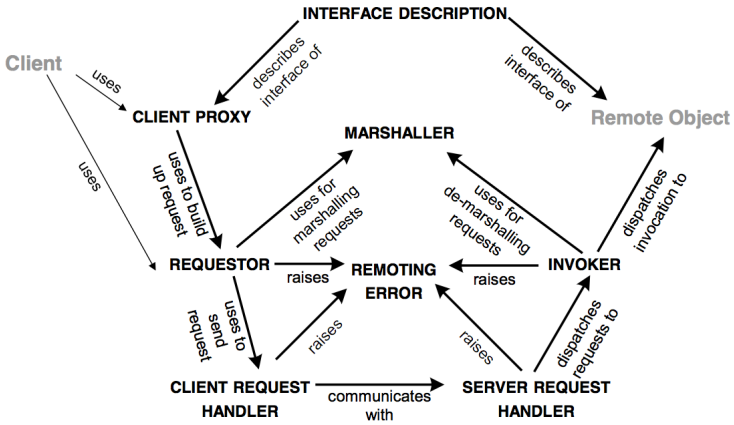
Server Application:

- Creates and configures remote object
- Activates remote object and makes it available to remote clients
- Publishes remote object in a naming service
- Destroys remote instances not needed anymore

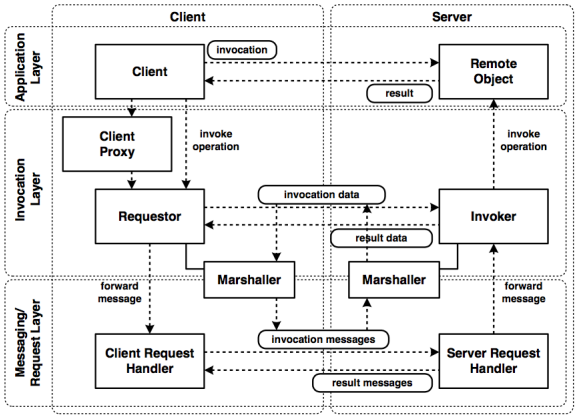


10.3 Basic Remoting Patterns

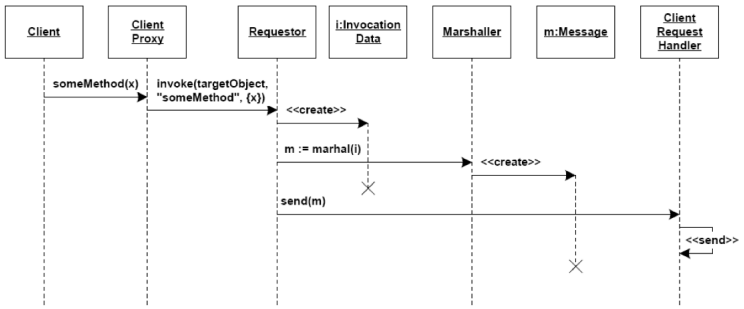
10.3.1 Overview



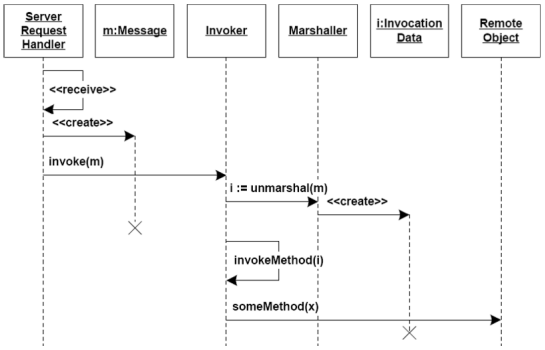
10.3.2 Interactions



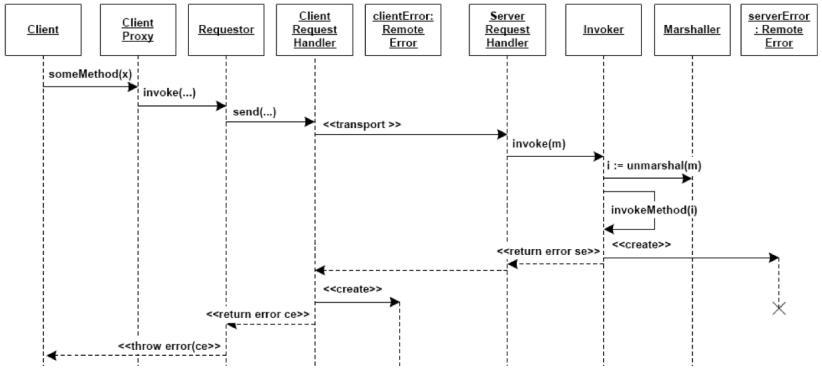
Basic invocation sequence on client side:



Basic invocation sequence on server side:

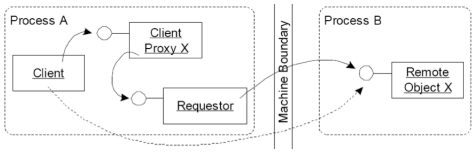


Remoting Error Propagation:



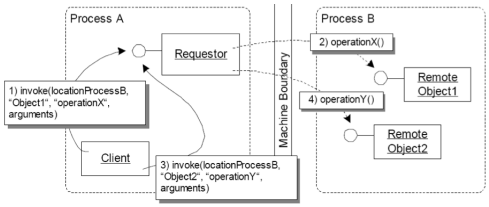
10.3.3 CLIENT PROXY

- Local object within the client process that offers the same interface as the remote object
- Simplifies the usage of the REQUESTOR
- Provides type safety on top of the REQUESTOR
- CLIENT PROXY is specific to the type of the remote object, typically generated from the INTERFACE DESCRIPTION
- Needs to be available on client side, either statically linked or dynamically loaded



10.3.4 REQUESTOR

- Constructs a remote invocation on the client side from parameters such as remote object location, remote object type, operation name and arguments and sends the invocation to the remote object
- Client should not need to perform these tasks itself

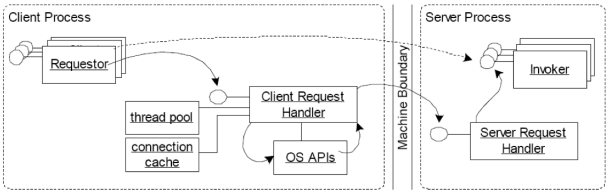


- Used by the client directly or by the client proxy object
- Tasks
  1. Accepts parameters from client (or from client proxy)
  2. Delegates task of marshalling the arguments to the MARSHALLER
  3. Sends the request using the CLIENT REQUEST HANDLER
  4. Throws REMOTING ERRORS in case of communication problems
  5. Delegates task of unmarshalling results to the MARSHALLER
  6. Returns result
- Independent of the specific remote object details (such as type, etc)
- Concurrent access to REQUESTOR needs to be synchronized

10.3.5 CLIENT REQUEST HANDLER

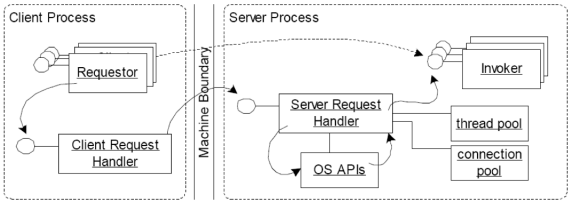
- Responsible for handling network communication in client application
  - Opening and closing network connections to server applications
  - Sending requests and receiving replies
  - Dispatching replies back to appropriate REQUESTOR (in case of asynchronous invocations)
  - Handling of timeouts and threading issues
  - Handling of invocation errors

- Connection handling
  Connections to a particular server / port may be shared with other invocations
- Typically shared between multiple REQUESTORS
- Tasks
  - Sending of requests
  - Receiving and dispatching of responses
  - Handling of timeouts, threading issues, and invocation errors
- Deals with all the communication issues of a server application
  - Receives messages from the network
  - Combines the message fragments to complete messages
  - Dispatches the messages to the correct INVOKER for further processing



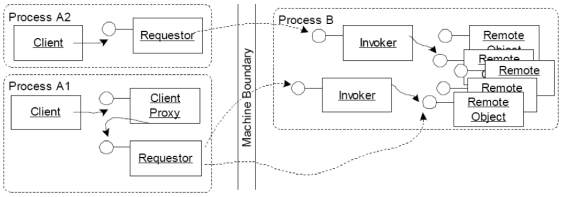
10.3.6 SERVER REQUEST HANDLER

- Deals with all the communication issues of a server application
  - Receives messages from the network
  - Combines the message fragments to complete messages
  - Dispatches the messages to the correct INVOKER for further processing
- Deals with messages, whereas the INVOKER deals with requests
- Must contact the correct INVOKER, i.e. message must contain information about the INVOKER as e.g. a name or ID
- Must be able to handle concurrent requests efficiently
  - One thread per connection
  - Connection-independent thread pool
- Connection establishment
  - New connection for each invocation
  - Reuse of existing connections (open connections consume resources)
- Tasks
  1. Receives data from network and combines them to a complete message
  2. Dispatches the message to the correct INVOKER (uses MARSHALLER to determine the correct INVOKER)
  3. Manages the required resources (connections, threads, ...)



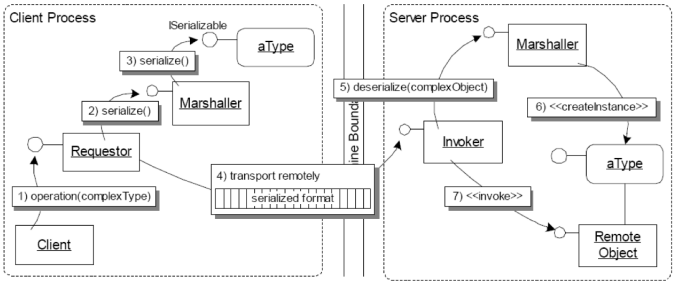
10.3.7 INVOKER

- Dispatches remote invocations from REQUESTORS to the relevant remote object using the received invocation information
  - Static dispatch: generated skeletons
  - Dynamic dispatch: reflection and dynamic lookup table
- Remote object might not be available all the time and only be activated on demand, which requires some part of the system to accept invocations and trigger the (re-)activation of the target remote object
- On server side, one or several INVOKER may exist (depends on CONFIGURATION GROUPS)
- Low-level network communications are handled by SERVER REQUEST HANDLER
- More than one servant might be used to implement a single remote object at runtime (⇒ POOLING) or might have to be activated (⇒ LIFECYCLE MANAGER)
- Tasks
  1. Accepts parameters from SERVER REQUEST HANDLER
  2. Delegates task of unmarshalling the arguments to the MARSHALLER
  3. Sends the request using the remote object (or its servant)
  4. Delegates task of marshalling results to the MARSHALLER
  5. Returns result to SERVER REQUEST HANDLER



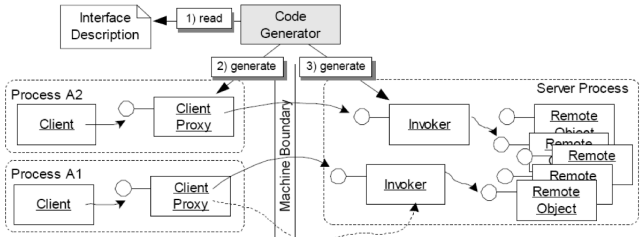
10.3.8 MARSHALLER

- Used to serialize invocation information into a byte stream (or into XML)
- Must be able to handle Cyclic structures, Alias references and Remote references
- Conversion from/to byte stream should be defined only once per type
- Hooks might be supported to provide custom marshallers



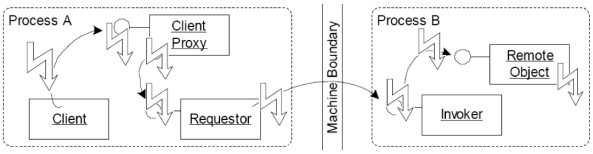
10.3.9 INTERFACE DESCRIPTION

- Describes the interface of a remote object
- Serves as the contract between CLIENT PROXY and INVOKER.
- Client and Server use either code generation or runtime configuration techniques to meet the interface contract
- Allows that client and server code are written in different programming languages



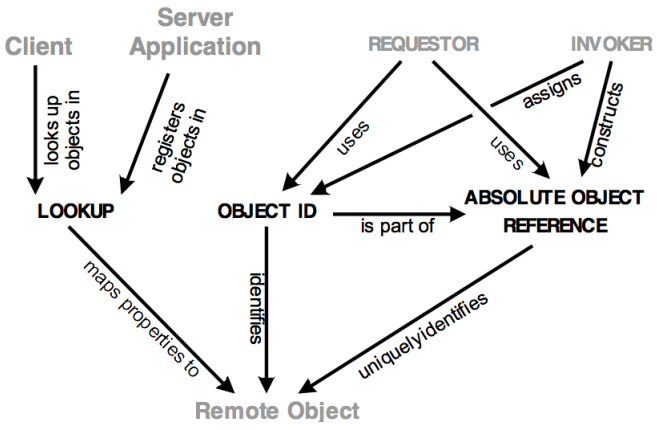
10.3.10 REMOTING ERROR

- Used to report problems that occur during a remote invocation, client must be able to distinguish between Distribution/communication related errors and Application-logic related errors
- Remoting errors detected inside the server application need to be transported back to the client
- Client might handle errors transparently (e.g. invoking another server)



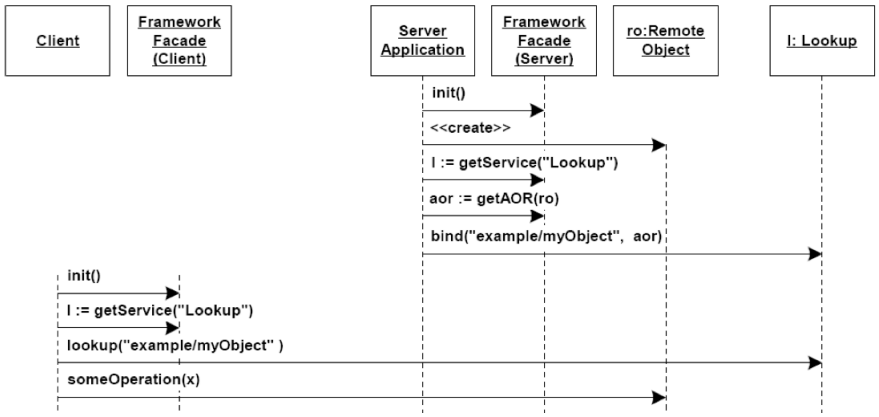
10.4 Identification Patterns

10.4.1 Overview

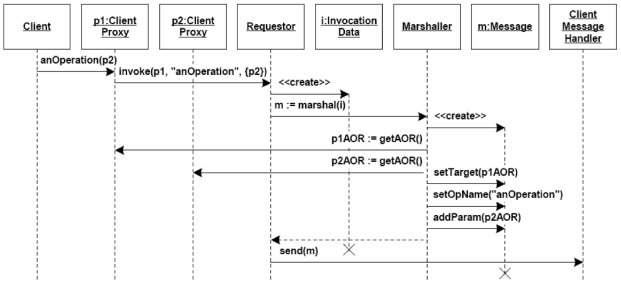


10.4.2 Interactions

Registration in Lookup:

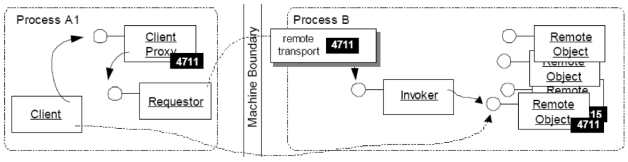


Marshalling of absolute object references: p1.anOperation(p2):



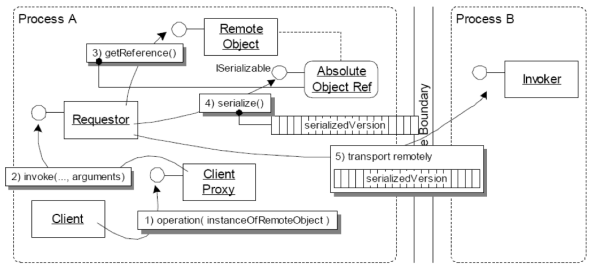
10.4.3 OBJECT ID

Refer to remote objects, rather than the servants realizing the remote object at runtime



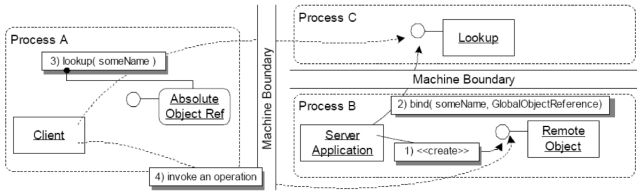
10.4.4 ABSOLUTE OBJECT REFERENCE

- Contains the information necessary for a client to connect the INVOKER
- Uniquely identifies invoker and remote object, typically contains Endpoint information, ID of the INVOKER Object ID
- Used by clients to exchange references to remote objects
- If several protocols are supported by the server, the protocol to be used by the client has to be included in the reference
- May be transient or persistent
  - Transient: only valid as long as server has not been restarted
  - Persistent: valid even after restart of the server application
- Client invokes an operation with a reference to another remote object



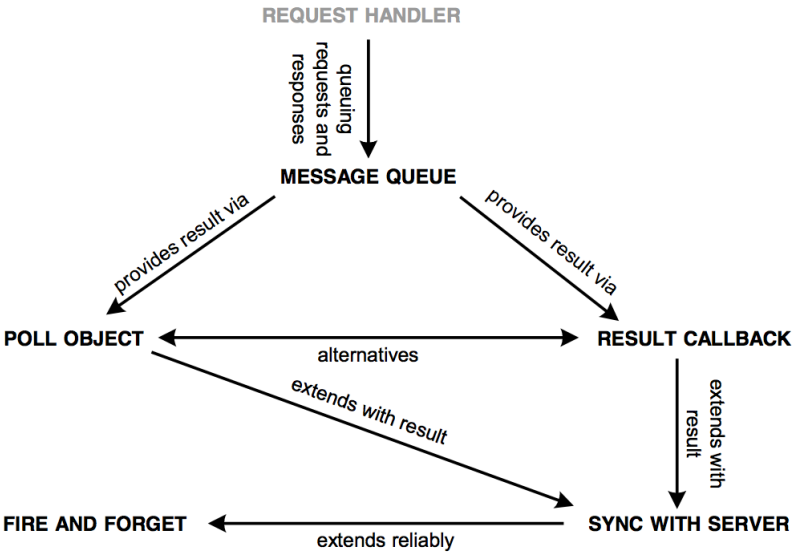
10.4.5 LOOKUP

- Provides ABOLUE OBJECT REFERENCES to remote objects
- Associates object references with properties (typically names)
- Used by clients to query for object references based on properties
- Provides an interface to bind and lookup object references
- Implemented as a remote object itself
- Only factory objects should be registered in the LOOKUP



10.5 Invocation Asynchrony Patterns

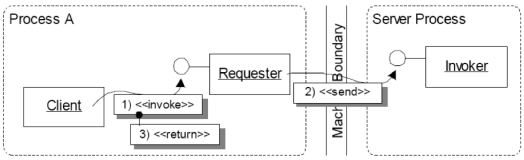
10.5.1 Overview



10.5.2 FIRE AND FORGET

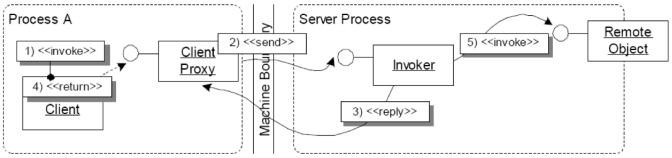
- Requestor sends the invocation across the network and returns control immediately to the caller
- Applicable if

- Client does neither expect a return value (void method) nor an exception
- Client simply needs to notify remote object of an event
- Reliability of the invocation is not critical (client does not get an acknowledgement from remote object)



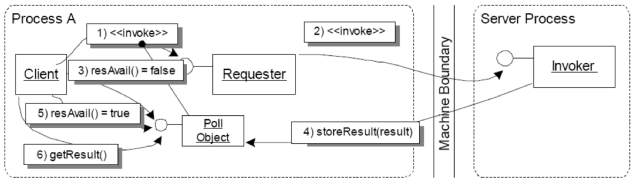
10.5.3 SYNC WITH SERVER

- Requestor sends the invocation across the network and waits for a reply from the server acknowledging the successful receipt of the invocation
- Acknowledgement is returned before operation is performed on server
- Applicable if
  - Client does neither expect a return value (void method) nor an exception
  - Fire-and-forget is too unreliable



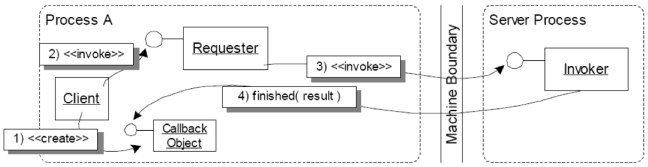
10.5.4 POLL OBJECT (Future)

- Requestor returns poll object immediately which is used by client to query result (client may block on poll object until result is available)
- Client can continue with other tasks asynchronously as long as result is not available on poll object
- Applicable if Client does not need result immediately to continue its execution



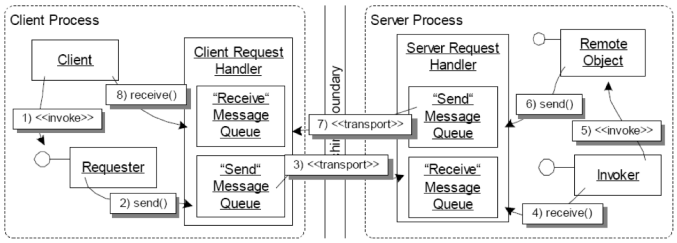
10.5.5 RESULT CALLBACK

- Provides a callback-based interface for remote invocations on the client
  - Client passes result-callback to the requestor
  - Invocation returns immediately after sending the invocation
  - Callback method is invoked on result-callback when the result is available
- Client is informed immediately when the result becomes available



10.5.6 MESSAGE QUEUE

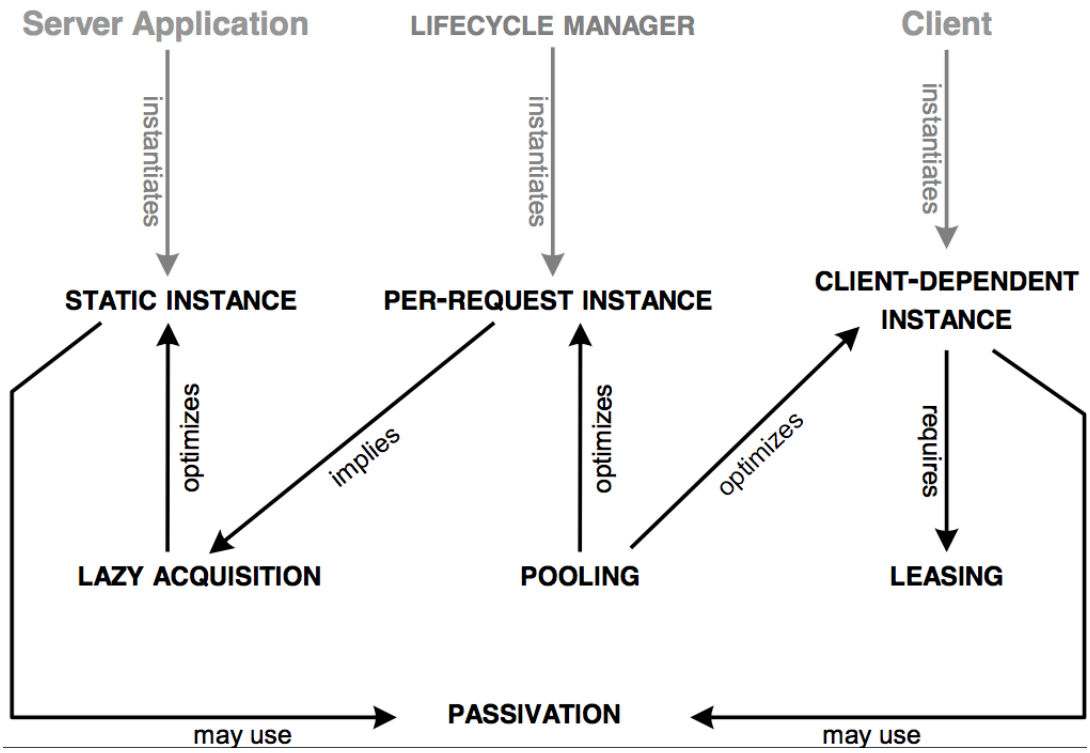
- Guarantees invocation and result delivery order
- Queues local to client/server request handler (no external message queue)



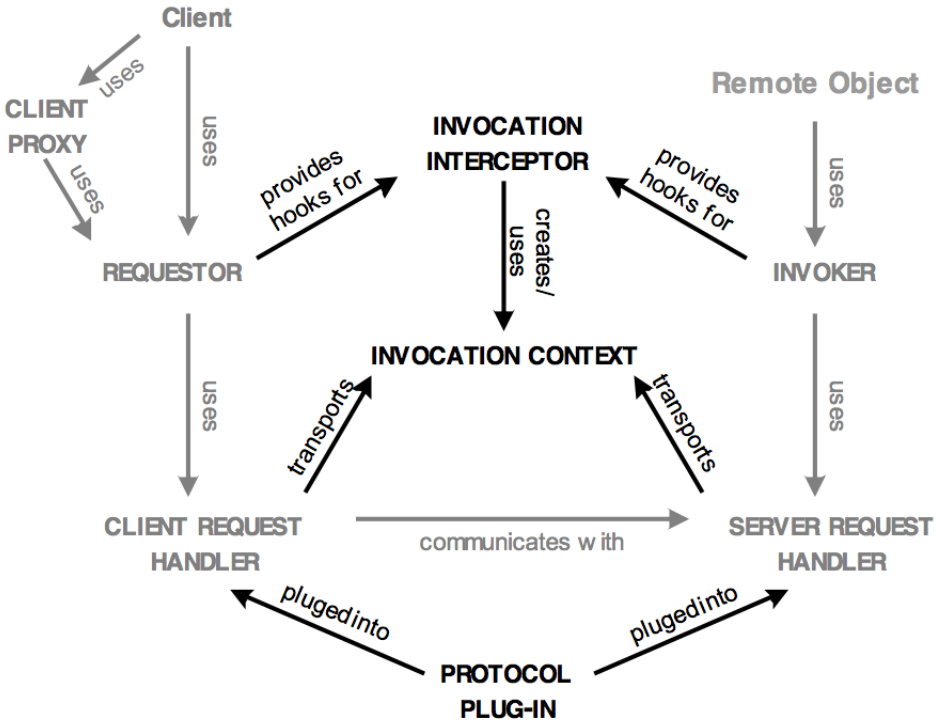
10.5.7 Relation among invocation asynchrony patterns

	Acknowledge-ment to client	Result to client	Responsibility for result
FIRE AND FORGET	No	No	-
SYNC WITH SERVER	Yes	No	-
POLL OBJECT	Yes	Yes	Client gets the result
RESULT CALLBACK	Yes	Yes	Client is informed via callback
MESSAGE QUEUE	Yes	Yes	Sync, poll object or callback

10.6 Lifecycle Management Patterns



10.7 Extension Patterns



11 Code Samples

11.1 Socket

Listing 65: Client

```
public class SocketBankDriver implements RemoteDriver {
  private RemoteBank bank = null;
  private Socket socket;
  private PrintWriter out;
  private BufferedReader in;
  public void connect(String[] args) throws IOException {
    socket = new Socket(args[0], Integer.valueOf(args[1]));
    bank = new RemoteBank(this);
    out = new PrintWriter(socket.getOutputStream());
    in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
  }
  public void disconnect() throws IOException{
    sendCommand("disconnect");
    socket.close(); bank = null;
  }
  public bank.Bank getBank(){ return bank; }
  public String sendCommand(String command) throws IOException {
    return sendCommand(command, "");
  }
  public String  sendCommand(String command, String param) throws IOException {
    out.println(command + ":" + param);
    out.flush();
    String input = in.readLine();
    String result = "";
    if (input != null) {
      String[] temp = input.split(":");
      result = (temp.length > 1 ? temp[1] : null );
    }
    return result;
  }
}
```

Listing 66: Server

```
public class SocketBankServer {
  private final int port;
  private MyBank bank;

  public SocketBankServer(int p) {
    port = p; bank = new MyBank();
  }
  public void start() {
    try (ServerSocket server = new ServerSocket(port)) {
      while (true) {
        Socket s = server.accept();
        Thread t = new Thread(new SocketHandler(s, bank));
        t.start();
      }
    } catch (IOException e) { System.err.println(e.getMessage()); }
  }
  public static void main (String args[]) {
    SocketBankServer server = new SocketBankServer(Integer.valueOf(args[0]));
    server.start();
  }
}
public class SocketHandler implements RequestHandler, Runnable {
  private final Socket socket;
  private CommandHandler cHandler;
  private boolean running = false;
  public SocketHandler(Socket s, MyBank b) {
    socket = s;
    cHandler = new CommandHandler(b, this);
  }
```

```
    public void run() {
      running = true;
      System.out.println("handle connection from " + socket);
      try {
        while (running) {
          Request req = receiveResult();
          String result = cHandler.handleCommand(req.getCommand(), req.getParam());
          if (!req.getCommand().equals("disconnect")) sendResponse(req.getCommand(), result);
        }
      } catch (IOException e) { }
          finally {
        try { socket.close();
        } catch (IOException e) { }
      }
    }
    public void stop() throws IOException {
      socket.close(); running = false;
    }
    public void sendResponse(String command, String param) throws IOException {
      PrintWriter out = new PrintWriter(socket.getOutputStream());
      System.out.println("Server send: '"+command+":"+param+"'");
      out.println(command + ":" + param);
      out.flush();
    }
    public Request receiveResult() throws IOException {
      BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
      String input = in.readLine();
      System.out.println("Server receive: '"+input+"'");
      String[] temp = input.split(":");
      return new Request(temp[0],temp.length > 1 ? temp[1] : "");
    }
  }
```

11.2 Internet

Listing 67: Client

```
  public class HttpBankDriver implements RemoteDriver {
    private String address;
    private RemoteBank bank = null;
    private URL url = null;
    public void connect(String[] args) throws IOException {
      bank = new RemoteBank(this);
      address = args[0];
      System.out.println(address);
    }
    public void disconnect() throws IOException {
      sendCommand("disconnect");
    }
    public Bank getBank() { return bank; }
    public String sendCommand(String command) throws IOException {
      return sendCommand(command, "");
    }
    public String sendCommand(String command, String param) throws IOException {
      url = new URL(address);
      HttpURLConnection c = (HttpURLConnection) url.openConnection();
      c.setRequestProperty("User-Agent", "SocketBank/HTTPBankDriver");
      c.setRequestMethod("GET");
      c.setUseCaches (false);
        c.setDoInput(true);
        c.setDoOutput(true);
        //Send request
        DataOutputStream wr = new DataOutputStream(c.getOutputStream ());
        wr.writeBytes("cmd="+command+"&param="+param);
        wr.flush ();
        wr.close ();
        BufferedReader r = new BufferedReader(new InputStreamReader(c.getInputStream()));
        String line;
      StringBuffer response = new StringBuffer();
        while((line = r.readLine()) != null) { response.append(line); }
```

```
34        r.close();
          String input = response.toString();
        String[] temp = input.split(":");
        return (temp.length > 1 ? temp[1] : null );
      }
39  }
```

Listing 68: Server

```
1   public class HttpBankServer {
        private final int port;
      private MyBank bank;
      public HttpBankServer(int p) {
        port = p;
6       bank = new MyBank();
      }
      public void start() throws IOException {
        HttpServer server = HttpServer.create(new InetSocketAddress(port), 0);
        server.createContext("/bank", new HttpRequestHandler(bank)).getFilters().add(new
            ParameterParser());
11      server.start();
      }
        public static void main(String[] args) throws IOException {
        HttpBankServer server = new HttpBankServer(Integer.valueOf(args[0]));
        server.start();
16    }
    }
    public class HttpRequestHandler implements HttpHandler, RequestHandler {
      private CommandHandler cHandler;
      private HttpExchange exchange;
21    public HttpRequestHandler(MyBank b) {
        cHandler = new CommandHandler(b, this);
      }
      public void handle(HttpExchange httpExchange) throws IOException {
        exchange = httpExchange;
26      exchange.getResponseHeaders().add("Content-type", "text/html");

        Map<String, Object> params = (Map<String, Object>) httpExchange.getAttribute("
            parameters");
        String cmd = (String) params.get("cmd");
        String param = (String) params.get("param");
31      String result = cHandler.handleCommand(cmd, param);
        String response =  cmd + ":" + result;
        exchange.sendResponseHeaders(200, response.length());
        OutputStream os = exchange.getResponseBody();
        os.write(response.getBytes());
36      os.close();
      }
    }
```

11.3 XmlRpc

Listing 69: Client

```
1   public class XmlRpcBankDriver implements RemoteDriver {
      private RemoteBank bank = null;
      private XmlRpcClient client = null;
      public void connect(String[] args) throws IOException {
        bank = new RemoteBank(this);
6       String address = args[0];
        XmlRpcClientConfigImpl config = new XmlRpcClientConfigImpl();
        config.setServerURL(new URL(address));
        client = new XmlRpcClient();
        client.setConfig(config);
11    }
      public void disconnect() throws IOException {
        sendCommand("disconnect");
      }
```

```
    public Bank getBank() { return bank; }
16  public String sendCommand(String command) throws IOException {
      return sendCommand(command, "");
    }
    public String sendCommand(String command, String param) throws IOException {
      System.out.println("Client send: '"+command+":"+param+"'");
21    List<Object> params = new ArrayList<Object>();
      params.add(command);
      params.add(param);
      String result;
      try {
26      result = (String) client.execute("Bank.handle", params );
        System.out.println("Client receive: '"+result+"'");
      } catch (XmlRpcException e) { }
      return (result.length() > 0 ? result : null );
    }
31
  }
```

Listing 70: Server

```
  public class XmlRpcBankServer {
    private final int port;
3   private static MyBank bank;
    public XmlRpcBankServer(int p) {
      port = p;
      bank = new MyBank();
    }
8   public static MyBank getBank() { return bank; }
    public void start() throws XmlRpcException, IOException {
      WebServer server = new WebServer(port);
      XmlRpcServer xmlRpcServer = server.getXmlRpcServer();
      PropertyHandlerMapping phm = new PropertyHandlerMapping();
13    phm.addHandler("Bank", ch.fhnw.jfmk.bank.server.handler.XmlRpcHandler.class);
      xmlRpcServer.setHandlerMapping(phm);
          server.start();
    }
    public static void main(String[] args) throws XmlRpcException, IOException {
18    XmlRpcBankServer server = new XmlRpcBankServer(Integer.valueOf(args[0]));
      server.start();
    }
  }
  public class XmlRpcHandler implements RequestHandler{
23  private CommandHandler cHandler;
    public XmlRpcHandler () {
      cHandler = new CommandHandler(XmlRpcBankServer.getBank(), this);
    }
    public String handle(String command, String param) throws IOException {
28    return cHandler.handleCommand(command, param);;
    }
  }
```

11.4 REST

Listing 71: Client

```
  public class RestBankDriver implements RemoteDriver {
    private RemoteBank bank;
    private Client client;
    private String address;
5   private WebResource res;
    public void connect(String[] args) throws IOException {
      bank = new RemoteBank(this);
      client = Client.create();
      address = args[0];
10    if (!address.endsWith("/")) address += "/";
    }
    public void disconnect() throws IOException {
```

```
    if (client != null) client.destroy();
  }
  public Bank getBank() { return bank; }
  public String sendCommand(String command) throws IOException {
    return sendCommand(command, "");
  }
  public String sendCommand(String command, String param) throws IOException {
    System.out.println("Client send: '" + command + ":" + param + "'");
    String mime = "application/plain", result = "";
    MultivaluedMap<String, String> formData = new MultivaluedMapImpl();
    String[] params = param.split(";");
    switch (command) {
    case "createAccount":
      res = client.resource(address + "accounts/create");
      formData.add("owner", param);
      result = res.accept(mime).post(String.class, formData);
      break;
    case "closeAccount":
      res = client.resource(address + "accounts/close/" + param);
      result = res.accept(mime).delete(String.class);
      break;
    case "getOwner":
      res = client.resource(address + "accounts/owner/" + param);
      result = res.accept(mime).get(String.class);
      break;
    case "getAccountNumbers":
      res = client.resource(address + "accounts");
      result = res.accept(mime).get(String.class);
      break;
    case "isActive":
      res = client.resource(address + "accounts/status/" + param);
      result = res.accept(mime).get(String.class);
      break;
    case "deposit":
      res = client.resource(address + "accounts/deposit/" + params[0]);
      formData.add("value", params[1]);
      result = res.accept(mime).put(String.class, formData);
      break;
    case "withdraw":
      res = client.resource(address + "accounts/withdraw/" + params[0]);
      formData.add("value", params[1]);
      result = res.accept(mime).put(String.class, formData);
      break;
    case "getBalance":
      res = client.resource(address + "accounts/balance/" + param);
      result = res.accept(mime).get(String.class);
      break;
    case "disconnect":
      bank = null;
      break;
    default:
      result = "unknown command";
      break;
    }
    return (result.length() > 0) ? result : null;
  }
}
```

Listing 72: Server

```
public class RestBankServer {
  private final int port;
  private HttpServer server;
  public RestBankServer(int p) throws IllegalArgumentException, NullPointerException,
      IOException {
    port = p;
    ResourceConfig rc = new ApplicationAdapter(new BankApplication(new MyBank()));
    server = GrizzlyServerFactory.createHttpServer("http://localhost:"+port, rc);
  }
  public void start() throws IOException {
```

```
    server.start();
    System.in.read();
    server.stop();
  }
  public static void main(String[] args) throws IOException {
    RestBankServer server = new RestBankServer(Integer.valueOf(args[0]));
    server.start();
  }
  public class BankApplication extends Application {
    private Set<Object> singletons = new HashSet<Object>();
    private Set<Class<?>> classes = new HashSet<Class<?>>();
    public BankApplication(MyBank b) {
      singletons.add(new RestHandler(b));
    }
    public Set<Class<?>> getClasses() { return classes; }
    public Set<Object> getSingletons() { return singletons; }
  }
}

@Singleton @Path("/bank")
public class RestHandler implements RequestHandler {
  private CommandHandler cHandler;
  public RestHandler(MyBank b) {
    cHandler = new CommandHandler(b, this);
  }
    @POST @Path("/accounts/create") @Produces("application/plain")
  public String postCreateAccount(@FormParam("owner") String owner) throws IOException {
    return cHandler.handleCommand("createAccount", owner);
  }

  @DELETE @Path("/accounts/close/{id}") @Produces("application/plain")
  public String deleteCloseAccount(@PathParam("id") String id) throws IOException {
    return cHandler.handleCommand("closeAccount", id);
  }
  @GET @Path("/accounts/owner/{id}") @Produces("application/plain")
  public String getOwner(@PathParam("id") String id) throws IOException {
    return cHandler.handleCommand("getOwner", id);
  }
  @GET @Path("/accounts") @Produces("application/plain")
  public String getAccountNumbers() throws IOException {
    return cHandler.handleCommand("getAccountNumbers", "");
  }
  @GET @Path("/accounts/status/") @Produces("application/plain")
  public String getStatus() throws IOException {
    return "null";
  }
  @GET @Path("/accounts/status/{id}") @Produces("application/plain")
  public String getStatus(@PathParam("id") String id) throws IOException {
    return cHandler.handleCommand("isActive", id);
  }
  @PUT @Path("accounts/deposit/{id}") @Produces("application/plain")
  public String putDeposit(@PathParam("id") String id, @FormParam("value") String value)
      throws IOException {
    return cHandler.handleCommand("deposit", id + ";" + value);
  }
  @PUT @Path("accounts/withdraw/{id}") @Produces("application/plain")
  public String putWithdraw(@PathParam("id") String id, @FormParam("value") String value)
      throws IOException {
    return cHandler.handleCommand("withdraw", id + ";" + value);
  }
  @GET @Path("/accounts/balance/{id}") @Produces("application/plain")
  public String getBalance(@PathParam("id") String id) throws IOException {
    return cHandler.handleCommand("getBalance", id);
  }
}
```