

# A Brief Introduction to Middleware

**Dozent:** Prof. Dr. Michael Eichberg

**Kontakt:** [michael.eichberg@dhw.de](mailto:michael.eichberg@dhw.de), Raum 149B

**Version:** 1.0.1

---

**Folien:** [HTML] <https://delors.github.io/ds-introduction-to-middleware/folien.en.rst.html>  
[PDF] <https://delors.github.io/ds-introduction-to-middleware/folien.en.rst.html.pdf>

**Fehler melden:** <https://github.com/Delors/delors.github.io/issues>

# 1. Middleware - Basics

## What is Middleware?

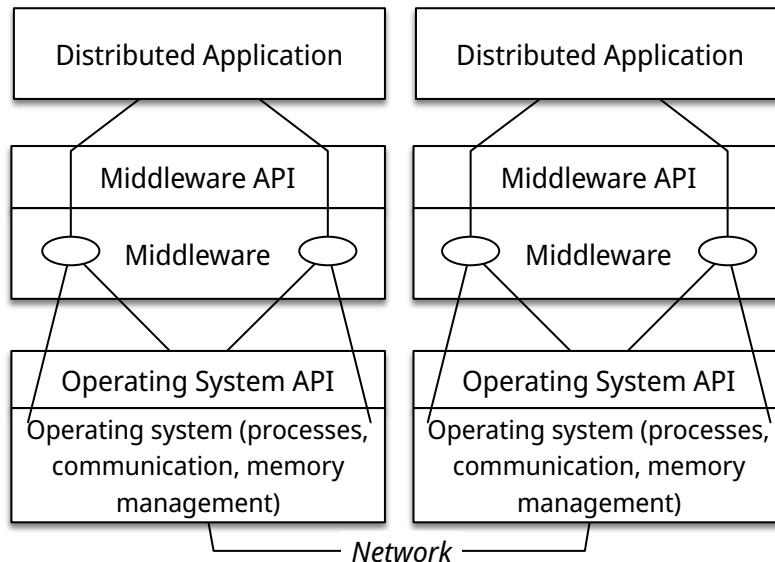


### Definition: Middleware

A class of software technologies that serve two purposes:

- I. reduce the complexity and
- II. manage the heterogeneity of distributed systems.

# Middleware as a Programming Abstraction



The programming abstractions offered by middleware hide some of the heterogeneity and manage some of the complexity that programmers of a distributed application have to deal with:

- ✓ Middleware always masks the heterogeneity of the underlying networks and hardware.
- ✓ Middleware usually masks the heterogeneity of operating systems and/or programming languages.
- ✓ *peripheral: Some middleware even masks the heterogeneity between implementations of the same middleware standard by different vendors.*

---

## Middleware ...

- A software layer above the operating system and below the application program that provides a common programming abstraction in a distributed system.
- A building block at a higher level than the APIs provided by the operating system (e.g. sockets)

## History

Old middleware standards - such as CORBA - were very complex and the implementations of different manufacturers were usually not fully compatible.

# Transparency Goals of Middleware from a Programming Perspective

Middleware provides transparency (when programming) in relation to one or more of the following dimensions:

- Location
- Concurrency
- replication
- Failures (but only to a limited degree)

## Assessment

Middleware is the software that makes a distributed system (DS) programmable.

# Middleware as Infrastructure

- Behind programming abstractions is a complex infrastructure that implements these abstractions  
Middleware platforms can be very complex software systems.
  - As the programming abstractions reach ever higher levels, the underlying infrastructure that implements the abstractions must grow accordingly.
  - Additional functionality is almost always implemented through additional software layers.
  - The additional software layers increase the scope and complexity of the infrastructure required to use the new abstractions.
- 

For decades, it has been observed that middleware has become increasingly complex, to the point where the complexity was barely manageable. At these points in time, new approaches were often developed that reduced the complexity until this in turn found its way into more complex middleware products.

Approaches such as REST have proven to be quite successful, but present developers with new challenges.

## Non-functional Requirements

The infrastructure takes care of non-functional properties that are normally ignored by data models, programming models and programming languages:

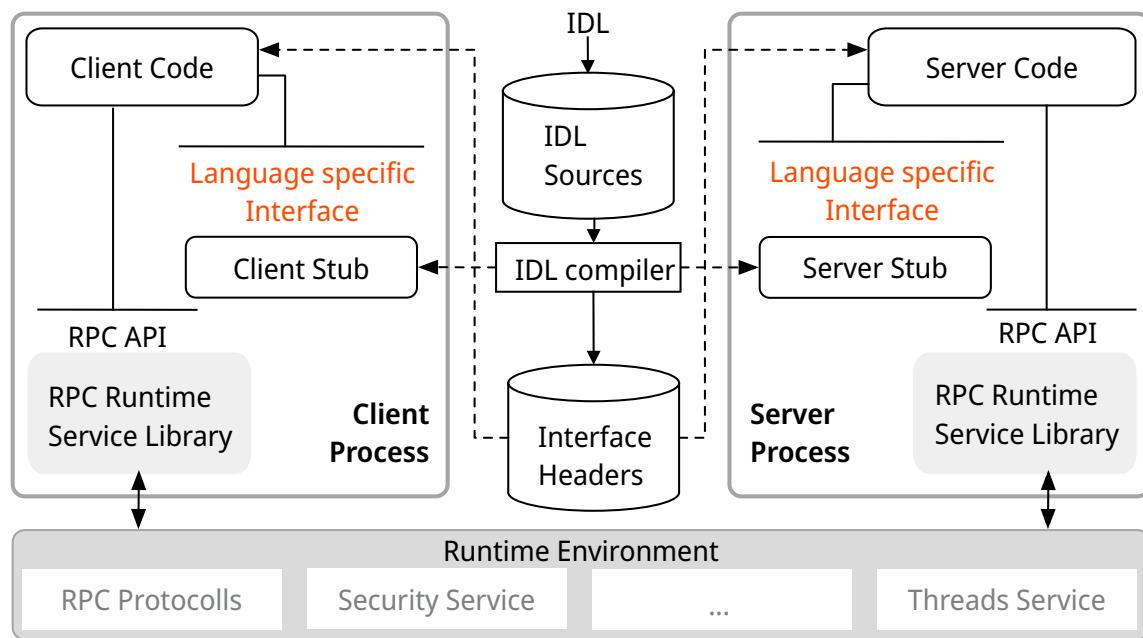
- Performance
- availability
- resource management
- reliability
- etc.

## Supporting Development and Operations

Middleware supports additional functions that make development, maintenance and monitoring easier and more cost-effective (excerpt):

- Logging
- Recovery
- (Programming) Language primitives for transactional delimitation  
(E.g., advanced transaction models (e.g. transactional RPC) or transactional file systems)

## Middleware - Conceptual Overview (historic)[1]



Today, the generation of stubs and skeletons - if required at all - is now typically carried out automatically by the middleware.

[1] Based on: Alonso; Web services: Concepts, Architectures and Applications; Springer, 2004

## Evolution of Middleware

- Middleware aims to hide the details of hardware, networks and distribution at a low level.
- Continuing trend towards ever more powerful primitives (*events*) that have additional properties or allow more flexible use of the concept.
- The development and appearance for the programmer is dictated by the trends in programming languages:
  - RPC and C
  - CORBA and C++
  - RMI (Corba) and Java
  - "Classic" web services and XML
  - RESTful web services and JSON

## 2. Middleware-Technologies

# Remote Procedure Call (RPC)

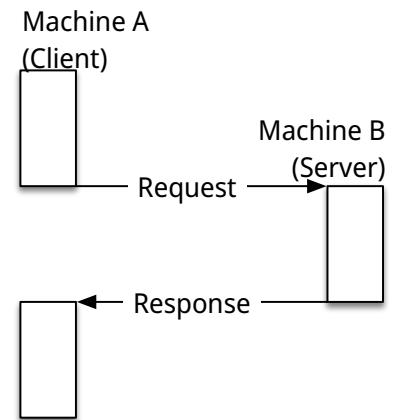
Focus: hiding network communication.

A process can call a procedure whose implementation is located on a remote computer:

- Distributed system programmers no longer have to worry about all the details of network programming (i.e. no more "explicit" sockets).
- Bridging the conceptual gap between calling local functionality via procedures and calling remote functionality via sockets.

## RPCs Conceptual (synchronous Communication)

- A server is a program that implements certain services.
- Clients want to use these services:
  - Communication takes place by sending messages (no shared memory, no shared disks, etc.)
  - Some minimal guarantees must be given (handling of errors, call semantics, etc.)



# RPCs - Key Issues and Challenges

## ? Question

Should remote calls be transparent or non-transparent for the developer?

### • Remark

A remote call is completely different from a local call; should the programmer be aware of this?

## ? Question

How can data be exchanged between machines that may use different representations for different data types?

Complex data types must be linearized:

**Marshalling:** the process of preparing the data into a form suitable for transmission in a message.

**Unmarshalling:** the process of restoring the data on arrival at its destination in order to obtain a faithful representation.

How do you find and bind the service you actually want in a potentially large collection of services and servers?

### • Remark

The aim is that the customer does not necessarily need to know where the server is located or even which server offers the service (location transparency).

How to deal with mistakes more or less elegantly:

- Server is down
- Communication is disrupted
- Server busy
- duplicate requests ...

# High-level View at RPC

## Observation

For programmers, a "remote" procedure call looks and works almost identically to a "local" procedure call - this is how transparency is achieved.

To achieve transparency, RPC introduced many concepts of middleware systems:

- *Interface Description Language (IDL)*
- Directory and naming services
- Dynamic binding
- Marshalling and unmarshalling
- Opaque references to refer to the same data structure or entity on the server for different calls.

(The server is responsible for providing these opaque references).

# RPC - Call Semantics

Suppose a client makes an RPC request to a service of a particular server. After the timeout expires, the client decides to resend the request. The final behavior depends on the semantics of the call (aka *Call Semantics*):

## Maybe (no guarantee)

The target method may have been executed and the response message(s) were lost or the method was not executed at all because the request was lost.

`XMLHttpRequests` and `fetch()` in web browsers use this semantics.

## At least once

The procedure will be executed as long as the server does not finally fail.

However, it is possible that it will be executed more than once if the client has resent the request after a timeout.

## At most once

The procedure is either executed once or not at all. Sending the request again does not result in the procedure being executed more than once.

## Exactly once

The system guarantees the same semantics as for local calls under the assumption that a crashed server will restart at some point.

Orphaned calls, i.e. Calls on crashed server computers are retained so that they can later be taken over by a new server.

## RPC - Assessment

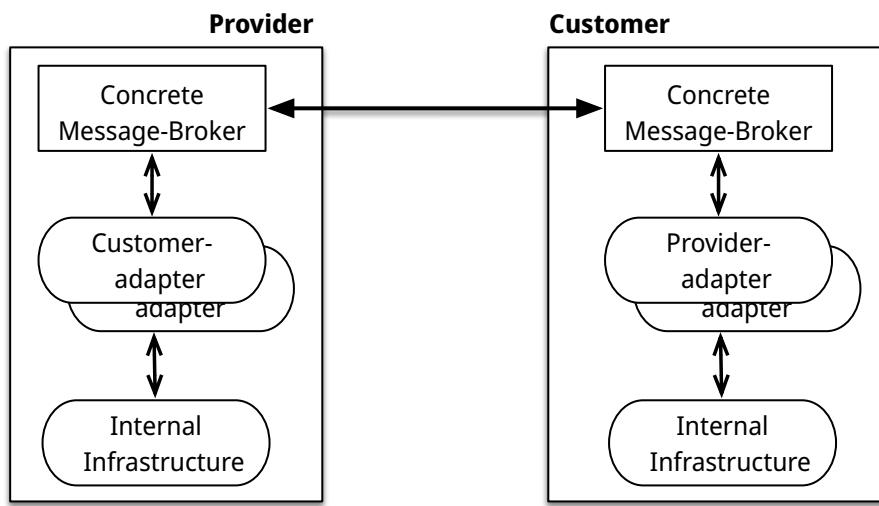
- ✓ RPC provides a mechanism to implement distributed applications in a simple and efficient way.
- ✓ RPC enables the modular and hierarchical structure of large distributed systems:
  - Client and server are separate entities
  - The server encapsulates and hides the details of the backend systems (such as databases)
- ! RPC is not a standard, but has been implemented in many different ways.
- ! RPC enables developers to set up distributed systems, but only solves selected aspects.

The Network File System (NFS) and SMB are two well-known applications based on RPC.

# Classic Web Services and SOAP

## Integration of Business Applications[2]

The problems of enabling cross-company point-to-point integration led to the development of the next generation of middleware technologies.



---

Each company uses its own "concrete" message broker(s) - if we want to communicate with multiple companies, we need to implement and maintain multiple adapters/solutions.

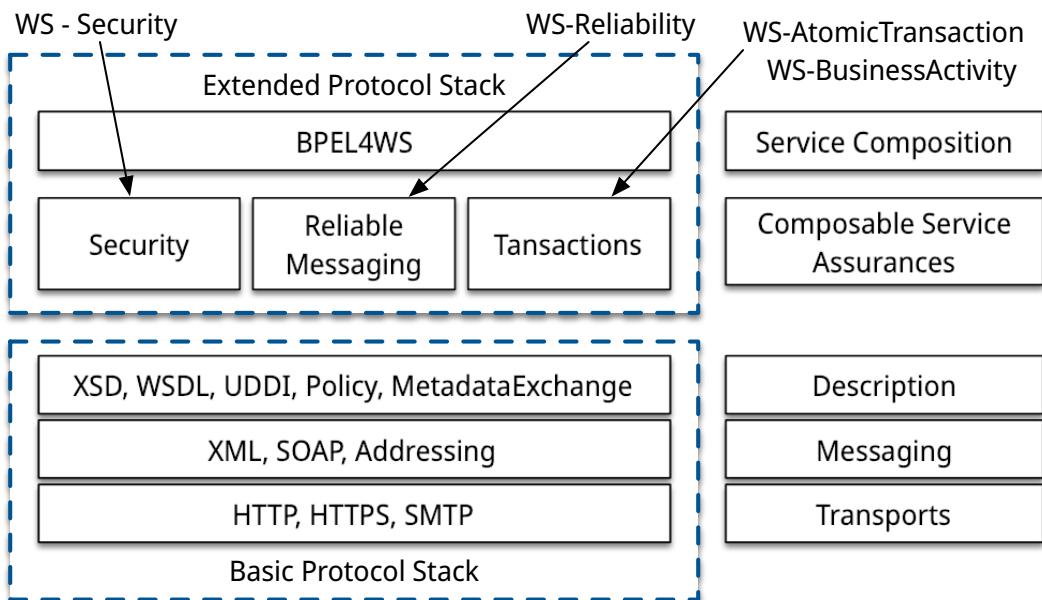
---

[2] Based on *Web Services - Concepts, Architectures and Applications*; Alonso et al.; Springer 2004

Webservices are self-contained, modular business applications that have open, internet-oriented, standards-based interfaces.

—UDDI Konsortium

# Web Services - Protocol Stack



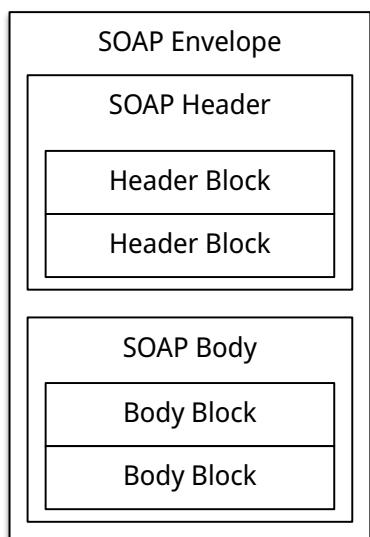
# SOAP

- SOAP is the protocol of classic web services and enables communication between applications.
  - SOAP comprises the following parts:
    - A message format that describes how a message can be wrapped in an XML document (envelopes, headers, body...)
    - A set of encoding rules for data
    - A description of how a SOAP message should be transported using the underlying transport protocol (HTTP or SMTP). How a SOAP message can be embedded in an HTTP request or in an e-mail (SMTP).
    - A set of rules to follow when processing a SOAP message and the entities involved in this processing; which parts of the messages should be read and by whom, and what action these entities should take if they do not understand the content.
- 

SOAP is a further development of XML-RPC and originally stood for Simple Object Access Protocol.

SOAP (from version 1.2) is a standard of the W3C.

# Structure of a SOAP message



Messages are envelopes in which the application's user data is enclosed.

A message has two main components:

**Header (optional):**

Intended for infrastructural data such as security or reliability.

**Body (mandatory):**

Intended for application level data. Each part can be divided into blocks.

## Example of a SOAP-Message

```
1 <SOAP-ENV:Envelope
2   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
3   SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" ↵
4
5   <SOAP-ENV:Header>
6     <t:Transaction xmlns:t="ws-transactions-URI" SOAP-ENV:mustUnderstand="1">
7       57539
8     </t:Transaction>
9   </SOAP-ENV:Header>
10
11  <SOAP-ENV:Body>
12    <m:GetLastTradePrice xmlns:m="Some-URI">
13      <symbol>DEF</symbol>
14    </m:GetLastTradePrice>
15  </SOAP-ENV:Body>
16
17 </SOAP-ENV:Envelope>
```

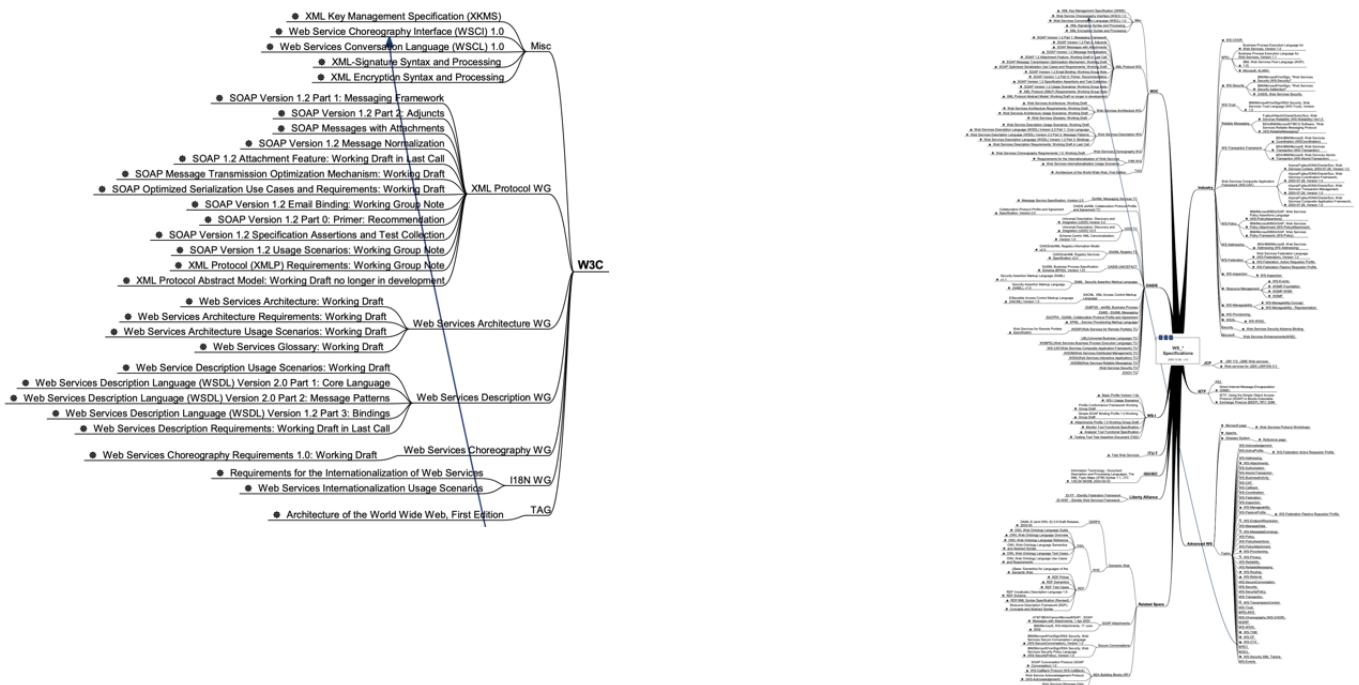
## Example of a SOAP-Request

```
1 POST /StockQuote HTTP/1.1
2 Host: www.stockquoteserver.com
3 Content-Type: text/xml; charset="utf-8"
4 Content-Length: nnnn
5 SOAPAction: "Some-URI"
6
7 <SOAP-ENV:Envelope
8   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
9   SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
10
11 <SOAP-ENV:Body>
12   <m:GetLastTradePrice xmlns:m="Some-URI">
13     <symbol>DIS</symbol>
14   </m:GetLastTradePrice>
15 </SOAP-ENV:Body>
16 </SOAP-ENV:Envelope>
```

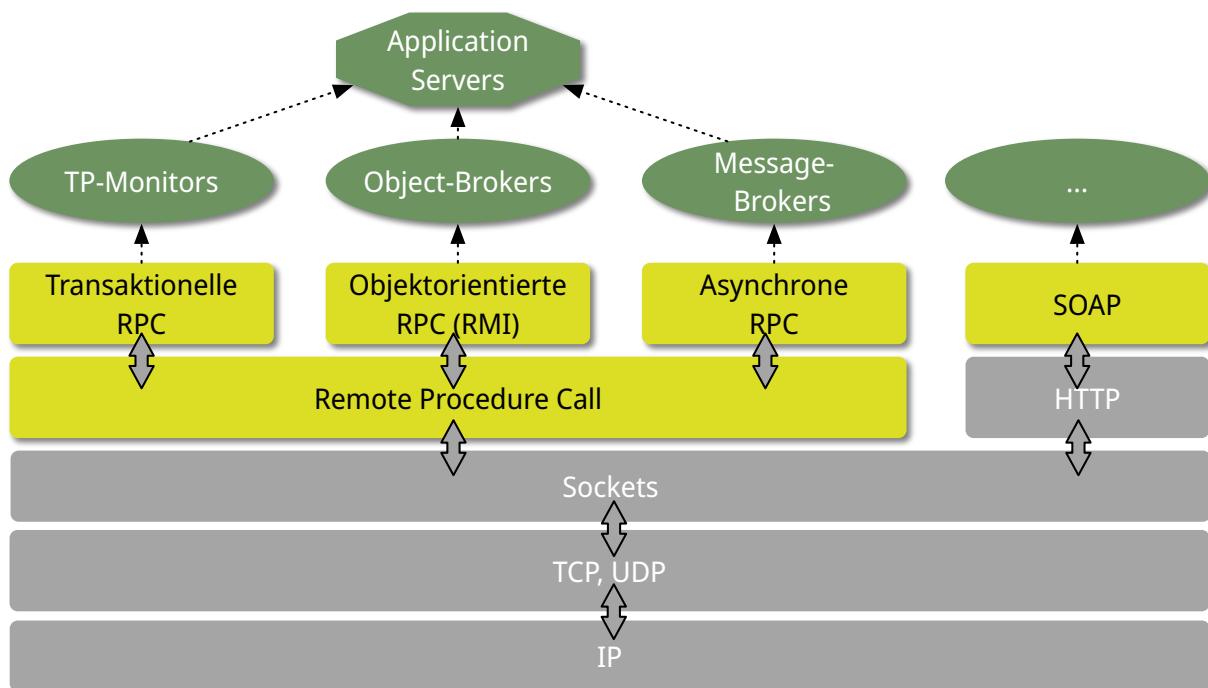
## Example of a SOAP-Response

```
1 | HTTP/1.1 200 OK
2 | Content-Type: text/xml; charset="utf-8"
3 | Content-Length: nnnn
4 |
5 | <SOAP-ENV:Envelope>
6 |   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
7 |   SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" ▶
8 |
9 | <SOAP-ENV:Body>
10 |   <m:GetLastTradePriceResponse xmlns:m="Some-URI">
11 |     <Price>34.5</Price>
12 |   </m:GetLastTradePriceResponse>
13 | </SOAP-ENV:Body>
14 | </SOAP-ENV:Envelope>
```

# Web Services - Standardization



# Overview



### 3. Messaging and Message-oriented Communication/Middleware

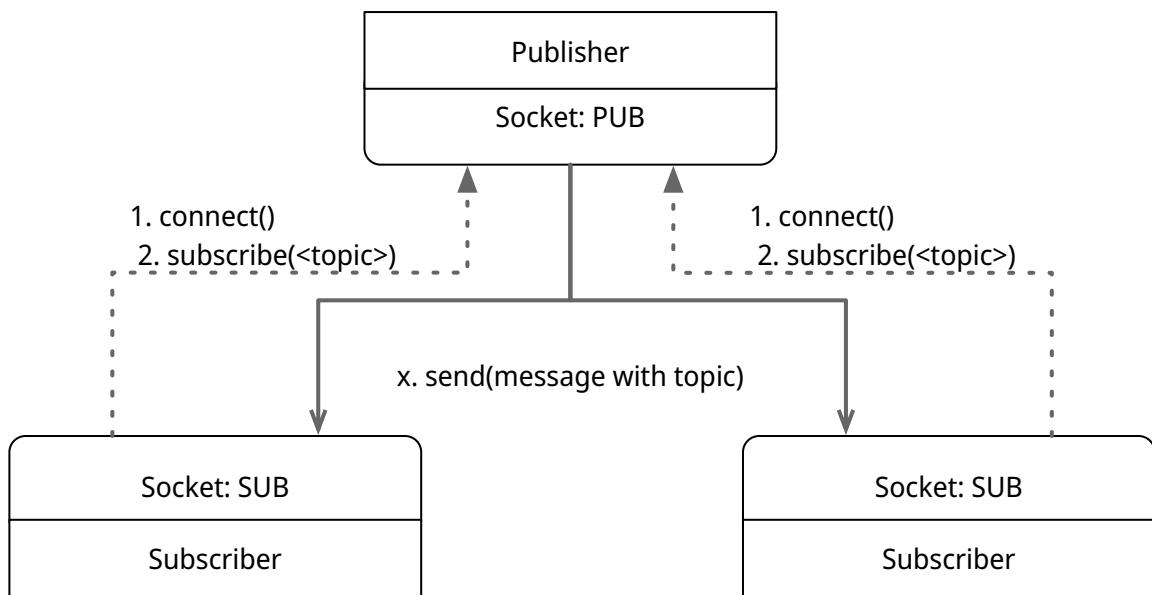
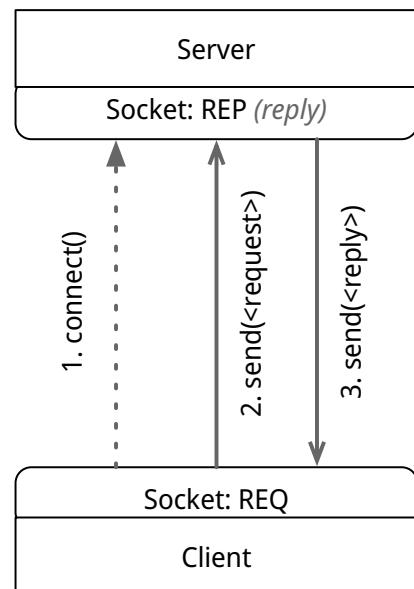
## ZeroMQ

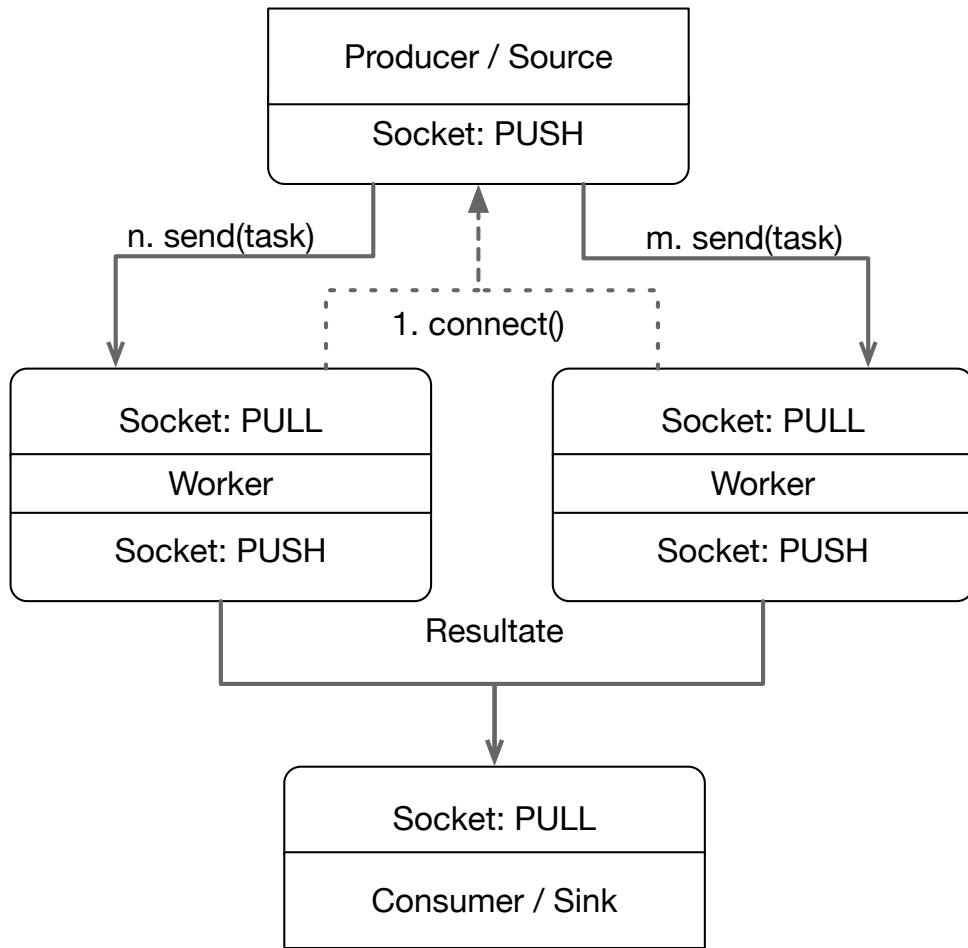
- ZeroMQ is a messaging infrastructure without an explicit server ("broker").
- ZeroMQ supports connection-oriented but asynchronous communication.
- ZeroMQ is based on classic sockets, but adds new abstractions to enable the following messaging patterns:
  - *request-reply*
  - *pub-sub* (publish-subscribe)
  - *pipeplining (processing in parallel)*
- ZeroMQ enables N-to-N communication.
- ZeroMQ supports many programming languages; the user is responsible for the appropriate marshalling or unmarshalling.

---

If, for example, the server is written in Java and the client in C, then the understanding of how a string is transferred may be different (e.g. terminated with `null` or provided with an explicit length).

# ZeroMQ - Messaging Patterns





**Client-Server:** Enables the "usual" communication between a client and a server. However, buffering may take place if the server is not available.

**Publish-Subscribe:** Allows clients to subscribe to a specific topic and then receive all messages published on that topic. A message with a specific topic is sent to all registered clients.

**Pipeline:** Enables a task to be sent to exactly one worker from a set of (homogeneous) workers.

## ZeroMQ - Example Publish-Subscribe: Publisher (Java)

```
1 import static java.lang.Thread.currentThread
2 import org.zeromq.SocketType;
3 import org.zeromq.ZMQ;
4 import org.zeromq.ZContext;
5
6 public class Publisher {
7     public static void main(String[] args) throws Exception {
8         try (ZContext context = new ZContext()) {
9             ZMQ.Socket publisher = context.createSocket(SocketType.PUB);
10            publisher.bind("tcp://*:5556");
11            publisher.bind("ipc://" + <endpoint>);
12
13            while (!currentThread().isInterrupted()) {
14                int zipcode = <some zipcode>
15                // Send to all subscribers
16                String update = String.format("%05d %s", zipcode, <some msg>);
17                publisher.send(update, 0);
18            } } } }
```

## ZeroMQ - Example Publish-Subscribe: Subscriber (Java)

```
1 import java.util.StringTokenizer;
2 import org.zeromq.SocketType;
3 import org.zeromq.ZMQ;
4 import org.zeromq.ZContext;
5
6 public class Subscriber{
7     public static void main(String[] args) {
8         try (ZContext context = new ZContext()) {
9             ZMQ.Socket subscriber = context.createSocket(SocketType.SUB);
10            subscriber.connect("tcp://localhost:5556");
11            subscriber.subscribe(<zipcode(Str)>.getBytes(ZMQ.CHARSET));
12            while(true) {
13                String string = subscriber.recvStr(0);
14                // e.g. take string apart:
15                //     part1: zipcode
16                //     part2: message
17                System.out.println(string);
18            } } } }
```

## ZeroMQ - Example Publish-Subscribe (Python)

```
1 import signal
2 import time
3 import zmq
4
5 signal.signal(signal.SIGINT,
6                 signal.SIG_DFL)
7
8 context = zmq.Context()
9 socket = context.socket(zmq.PUB)
10 socket.bind('tcp://*:5555')
11
12 for i in range(5):
13     socket.send(b'status 5')
14     socket.send(b'All is well')
15     time.sleep(1)
```

```
1 import signal
2 import zmq
3
4
5 signal.signal(signal.SIGINT,
6                 signal.SIG_DFL)
7
8 context = zmq.Context()
9 socket = context.socket(zmq.SUB)
10 socket.connect('tcp://localhost:5555')
11 socket.setsockopt(zmq.SUBSCRIBE, b'status')
12
13 while True:
14     message = socket.recv_multipart()
15     print(f'Received: {message}')
```

---

Bzgl. des Handlings von Signalen in Python siehe auch:

<https://docs.python.org/3/library/signal.html#signal.signal>

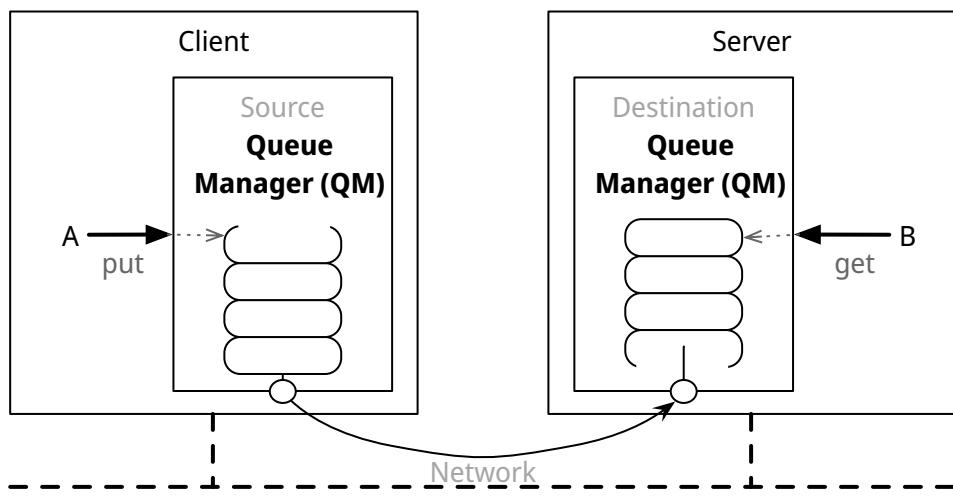
## MOM - Message Oriented Middleware

- MOM or message-queueing systems support persistent asynchronous communication.
- Very large messages are supported.
- There is only a guarantee that messages are ultimately placed in the recipient's queue and that the messages arrive in the correct order.  
(In particular, there is no guarantee that the message will be read).
- The sender and recipient are not necessarily active at the same time.
- Messages always have a unique recipient and virtually arbitrary content.

## MOM - Basic Interface

Operation	Description
PUT	Places a message in a specific queue.
GET	Blocks at a specific queue until a message is available. Removes the first message.
POLL	Checks whether a message is available in a specific queue. Removes the first message if necessary. POLL never blocks.
NOTIFY	Registers a handler ( <i>callback</i> ) that is called when a message is added to a specific queue.

# MOM - Queue Managers



*Queue managers* are the central building block of message queueing systems. In general, there is (at least conceptually) one local *Queue Manager* per process. A *Queue Manager* is a process that stores and manages messages in queues. If required, it can manage several queues and forward them to other *Queue Managers*.