public class Employee {
    public String name;
    public String address;
    public transient int SSN;
    public int number;
}
Chapter 4. Encoding and Evolution
Overview
Communication

- **Message Passing**
- OSI Model
- Socket-based Communication
- Message-oriented Middleware
- Service-oriented Middleware
- Database-oriented Middleware
Processes communicate
- With themselves
- With other processes on the same machine
- With processes on remote machines over the network
  - Data often needs to pass process boundaries!

Processes are heterogeneous
- Different languages, address spaces, access rights, hardware resources, complexities, interfaces, ...
  - Communication models/protocols needed!

Process communication is expensive
- Communication channels (buses, network, memory, ...) have limited speed, bandwidth and throughput
  - Number and size of messages matters!
Message Passing
Terminology

- A communication model that restricts the exchange of information between independent entities to the actions of sending and receiving of messages.
- **Entity**: Thread, process, subroutine, function, object, actor, ...
- **Message** (in some contexts “mail”):
  - Container for data that implies information or commands
  - Often carries metadata, e.g., size, receiver and sender information
  - Can have any format understood by sender and receiver
- **Buffer** (in some contexts “message queue” or “mailbox”):
  - Memory reserved by the communicating entities to store messages
  - (Usually) order messages by time of arrival and process them in order
  - A communication may involve multiple buffers (send buffers, system buffers, message queues, ...)

Messages can have any format understood by sender and receiver to enable replies.
Message Passing
Thread Messaging

Sender (Thread)

Receiver (Thread)

ReentrantLock messageMutex = new ReentrantLock();
Queue<Work> messageQueue = new ArrayDeque<Work>();

try {
    messageMutex.lock();
    messageQueue.offer(message);
} finally {
    messageMutex.unlock();
}

try {
    messageMutex.lock();
    message = messageQueue.poll();
} finally {
    messageMutex.unlock();
}

Does not work if sender and receiver live in different address spaces (i.e. processes)
Message Passing
Thread Messaging – Distributed Principle

**Sender** (Thread)

... send(receiver, message) ...

Message in local address space

**Receiver** (Thread)

... receive(sender, message) ...

Buffer space for incoming message in receiver address space

sender/receiver:
- Direct address: node ID, process ID
- Indirect address: mailbox, socket, channel, ...

**Distributed Data Management**
Communication

Thorsten Papenbrock
Slide 8
**Message Passing Middleware**

- **Sender** (Thread):
  - ... send(receiver, message) ...

- **Middleware** (Thread(s)):
  - A distributed software layer that supports the message transfer
  - Can provide additional buffers
  - Usually lives partially in both the sender and the receiver address space

- **Receiver** (Thread):
  - ... receive(sender, message) ...

---

**Distributed Data Management**

Communication

ThorstenPapenbrock
Slide 9
Message Passing

No Buffering vs. Buffering

No Buffering Communication

- The receiver allocates the memory for messages on demand.
- Requires rendezvous protocol on each message:
  1. Sender signals send attempt with the size of the message and waits.
  2. Receiver allocates memory when ready and signals readiness.
  3. Sender sends the message.

Buffering Communication

- The receiver maintains a pre-allocated buffer space for incoming messages.
- If the buffer is full, the sender blocks or drops the message.
Message Passing

Transient vs. Persistent

**Transient Communication**
- Message is copied directly from sender address space to receiver buffer.
- Both processes need to be active and memory for the message in the target buffer needs to be free.

**Persistent Communication**
- Message is first copied to a buffer inside the middleware and then send to the receiver.
- Middleware stores the message for as long as it takes to deliver it.
**Message Passing**

**Synchronous vs. Asynchronous**

### Synchronous (blocking) Communication
- **Send()** returns not until the data is copied out of its local message structure.
- **Receive()** returns when the message is fully received.
- After the returns:
  - The local messages/buffers can safely be modified.
  - Sender and receiver might know if the send succeeded.

### Asynchronous (non-blocking) Communication
- **Send()** returns directly, which is before the data is copied out of the local message structure.
- **Receive()** returns directly either with message or status code.
- After the returns:
  - The local messages/buffers should not be modified.
  - Only receiver knows if the send succeeded.

*Requires middleware to transmit the data!*
Middleware can support different levels of synchronization.

Middleware uses additional thread(s) that manage the message transfer.

Often also counts as asynchronous w.r.t. sender and receiver.

Message Passing
Synchronization Levels with Middleware

Sender
- send

Middleware

Receiver

Distributed Data Management
Communication

Thorsten Papenbrock
Slide 13
Overview

Communication

- Message Passing
- **OSI Model**
- Socket-based Communication
- Message-oriented Middleware
- Service-oriented Middleware
- Database-oriented Middleware
Distributed system communication is always based on low-level message passing.

Messages are serializes data with well defined (header) metadata.

Higher-level communication principles based on message passing, services, or databases are implemented on top of low-level message passing subsystems.

Message passing requires protocols for the exchange and routing of messages.
Communication Protocol

- Set of rules that govern the format, contents, and meaning of messages
- Provide standards for how to send and receive messages
- Is either connection-based (handshake before and after communication) or connectionless (no handshake, but simple message sends)

Open Systems Interconnection Reference Model (OSI Model)

- Layered model for network communication protocols
- Developed by the International Standards Organization (ISO) in 1983
### OSI Model

#### Communication Layers

<table>
<thead>
<tr>
<th>Layer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Physical</td>
<td>Standardization of the physical connection and representation of 0s and 1s</td>
</tr>
<tr>
<td>2 Data link</td>
<td>Means to detect/correct errors and keep sender/receiver in same pace</td>
</tr>
<tr>
<td>3 Network</td>
<td>Routing of messages in networks and handling message congestion via buffers</td>
</tr>
<tr>
<td>4 Transport</td>
<td>Support for reliable communication or real-time streaming features</td>
</tr>
<tr>
<td>5 Session</td>
<td>Support for sessions between applications</td>
</tr>
<tr>
<td>6 Presentation</td>
<td>Description of the logical format of a message’s serialized data</td>
</tr>
<tr>
<td>7 Application</td>
<td>Application-specific protocols, such as e-mail, Web access or file transfer</td>
</tr>
</tbody>
</table>

**Network**
OSI Model

Communication Formats

Node A -> Network -> Node B

- Physical
- Data link
- Network
- Transport
- Session
- Presentation
- Application

Message

Distributed Data Management
Communication

Thorsten Papenbrock
Slide 19
OSI Model

Intermediate Nodes

Node A

1. Physical
2. Data link
3. Network
4. Transport
5. Session
6. Presentation
7. Application

Intermediate Node

Node B

1. Physical
2. Data link
3. Network
4. Transport
5. Session
6. Presentation
7. Application

Distributed Data Management
Communication

ThorstenPapenbrock
Slide 20
OSI Model

Protocol Examples

1. **Physical layer**
   - EIA/TIA-232
   - EIA/TIA-449
   - ITU-T V-Series
   - I.430
   - I.431
   - PDH
   - SONET/SDH
   - PON
   - OTN
   - DSL
   - IEEE 802.3
   - IEEE 802.11
   - IEEE 802.15
   - IEEE 802.16
   - IEEE 1394
   - ITU-T G.hn PHY
   - USB
   - Bluetooth
   - RS-232
   - RS-449

2. **Data link layer**
   - ATM
   - ARP
   - IS-IS
   - SDLC
   - HDLC
   - CSLIP
   - SLIP
   - GFP
   - PLIP
   - IEEE 802.2
   - LLC
   - MAC
   - L2TP
   - IEEE 802.3
   - Frame Relay
   - ITU-T G.hn DLL
   - PPP
   - X.25 PLP
   - Q.922 LAPF

3. **Network layer**
   - IP (IPv4, IPv6)
   - ICMP
   - IPsec
   - IGMP
   - IPX
   - AppleTalk
   - X.25 PLP

4. **Transport layer**
   - TCP
   - UDP
   - SCTP
   - DCCP
   - SPX

5. **Session layer**
   - Named pipe
   - NetBIOS
   - SAP
   - PPTP
   - RTP
   - SOCKS
   - SPDY

6. **Presentation layer**
   - MIME
   - XDR
   - ASN.1
   - ASCII
   - PGP

7. **Application layer**
   - NNTP
   - SIP
   - SSL
   - DNS
   - FTP
   - Gopher
   - HTTP
   - NFS
   - NTP
   - SMPP
   - SMTP
   - SNMP
   - Telnet
   - DHCP
   - Netconf
   - more....

- Physical standards for network devices and interconnects
- E.g. volt specifications for 1 and 0, transmission rates, size and shape of connectors, ...
7. **Application layer**
   - NNTP, SIP, SSI, DNS, FTP, Gopher, HTTP, NFS, NTP, SMPP, SMTP, SNMP, Telnet, DHCP, Netconf, more...

6. **Presentation layer**
   - MIME, XDR, ASN.1, ASCII, PGP

5. **Session layer**
   - Named pipe, NetBIOS, SAP, PPTP, RTP, SOCKS, SPDY

4. **Transport layer**
   - TCP, UDP, SCTP, DCCP, SPX

3. **Network layer**
   - IP (IPv4, IPv6), ICMP, IPsec, IGMP, IPX, AppleTalk, X.25 PLP

2. **Data link layer**

1. **Physical layer**

- Sending of datagrams in local network
- Direct host-to-host messaging; no routing
- Addressing e.g. via MAC address
  - e.g. 34:f3:9a:fa:fb:59
- Hardware dependent
  - drivers needed
- Abstracting hardware details to above layers
- Packetizing, (local) addressing, transmission and receiving of data
OSI Model
Protocol Examples

7. **Application layer**
   - NTTP, SIP, SSL, DNS, FTP, Gopher, HTTP, NFS, NTP, SMPP, SMTP, SNMP
   - Telnet, DHCP, Netconf

6. **Presentation layer**
   - MIME, XDR, ASN.1, ASCII, PGP

5. **Session layer**
   - Named pipe, NetBIOS, SAP, PPTP, RTP, SOCKS, SPDY

4. **Transport layer**
   - TCP, UDP, SCTP, DCCP, SPX

3. **Network layer**
   - IP (IPv4, IPv6), ICMP, IPsec, IGMP, IPX, AppleTalk, X.25 PLP

2. **Data link layer**
   - ATM, ARP, IS-IS, SDLC, HDLC, CSLIP, SLIP, GPF, PLP, IEEE 802.2, LLC, MAC, L2TP, IEEE 802.3, Frame Relay
   - ITU-T G.hn DLL, PPP, X.25 LAPB, Q.922 LAPF

1. **Physical layer**
   - IEEE 802.3, IEEE 802.11, IEEE 802.15

- Routing of datagrams across networks
- Addressing e.g. via IP addresses
  - for addressing and routing
  - map to MAC addresses
  - e.g. 172.17.5.57
- Abstrating the actual network topology to above layers
- Packetizing, (global) addressing and routing of data
### OSI Model

#### Protocol Examples

**7. Application layer**
- NNTP, SIP, SSL, DNS, FTP, Gopher, HTTP, NFS, NTP, SMTP, SNMP, Telnet, DHCP, Netconf, more...

**6. Presentation layer**
- MIME, XDR, ASN.1, ASCII, PGP

**5. Session layer**
- Named pipe, NetBIOS, SAP, PPTP, RTP, SOCKS, SPDY

**4. Transport layer**
- TCP, UDP, SCTP, DCCP, SPX

**3. Network layer**
- IP (IPv4, IPv6), ICMP, IPsec, IGMP, IPX, AppleTalk, X.25 PLP

**2. Data link layer**

**1. Physical layer**

- Managing the datagram exchange
  - host-to-host (via arbitrary hops)
  - communication protocol
  - communication channel

- Port numbers for application addressing
  - e.g. 8080

- Abstracting communication details to above layers

- Packetizing of data

---

**Distributed Data Management**

Communication

---

**Slide 24**
OSI Model
Protocol Examples

TCP
- reliable (flow control)
- connection-based
- slow
- lost-message resents
- message ordering
- error correction
- duplicate removal
- congestion control

UDP
- unreliable
- connectionless
- fast

Managing the datagram exchange
- host-to-host (via arbitrary hops)
- communication protocol
- communication channel

Port numbers for application addressing
- e.g. 8080

Abstracting communication details to above layers

Packetizing of data

1. Physical layer
   - EIA/TIA-232
   - EIA/TIA-449
   - ITU-T V-Series
   - 1.430
   - 1.431
   - PDH
   - SONET/SDH
   - PON
   - OTN
   - DSL
   - IEEE 802.3
   - IEEE 802.11
   - IEEE 802.15
   - IEEE 802.16
   - IEEE 1394
   - ITU-T G.hn
   - PHY
   - USB
   - Bluetooth
   - RS-232
   - RS-449

2. Data link layer
   - ATM
   - ARP
   - IS-IS
   - SDLC
   - HDLC
   - CSLIP
   - SLIP
   - GFP
   - PLP
   - IEEE 802.2
   - LLC
   - MAC
   - L2TP
   - IEEE 802.3
   - Frame Relay
   - ITU-T G.hn
   - DLL
   - PPP
   - X.25 PLP
   - Q.922 LAPF

3. Network layer
   - IP (IPv4, IPv6)
   - ICMP
   - IPsec
   - IGMP
   - IPX
   - AppleTalk
   - X.25 PLP

4. Transport layer
   - TCP
   - UDP
   - SCTP
   - DCCP
   - SPX

5. Session layer
   - Named pipe
   - NetBIOS
   - SAP
   - PPTP
   - RTP
   - SOCKS
   - SPDY

6. Presentation layer
   - MIME
   - XDR
   - ASN.1
   - ASCII
   - PGP

7. Application layer
   - NNTP
   - SIP
   - SSI
   - DNS
   - FTP
   - Gopher
   - HTTP
   - NFS
   - NTP
   - SMPP
   - SMTP
   - SNMP
   - Telnet
   - DHCP
   - Netconf

Distributed Data Management
Communication

Slide 25
OSI Model

Protocol Examples

1. Physical layer
   - EIA/TIA-232 • EIA/TIA-449
   - ITU-T V-Series • I.430 • I.431 • PDH
   - SONET/SDH • PON • OTN • DSL
   - IEEE 802.3 • IEEE 802.11 • IEEE 802.15
   - IEEE 802.16 • IEEE 1394
   - ITU-T G.hn PHY • USB • Bluetooth
   - RS-232 • RS-449

2. Data link layer
   - ATM • ARP • IS-IS • SDLC • HDLC • CSLIP
   - SLIP • GFP • PLIP • IEEE 802.2 • LLC
   - MAC • L2TP • IEEE 802.3 • Frame Relay
   - ITU-T G.hn DLL • PPP • X.25 PLP
   - Q.922 LAPF

3. Network layer
   - IP (IPv4 • IPv6) • ICMP • IPsec • IGMP
   - IPX • AppleTalk • X.25 PLP

4. Transport layer
   - TCP • UDP • SCTP • DCCP • SPX

5. Session layer
   - Named pipe • NetBIOS • SAP • PPTP
   - RTP • SOCKS • SPDY

6. Presentation layer
   - MIME • XDR • ASN.1 • ASCII • PGP

7. Application layer
   - NNTP • SIP • SSL • DNS • FTP • Gopher
   - HTTP • NFS • NTP • SMPP • SMTP • SNMP
   - Telnet • DHCP • Netconf • more....

- Software layers
- Creation and interpretation of data
- Also: “Your application”
- Not all communication requires protocols from these layers

- Use the (reliable or unreliable) channels (identified by IP + Port) to send/receive data

Higher level communication protocols
- client-server
- peer-to-peer

“IP + Port + Application” is a service

Distributed Data Management
Communication
OSI Model
Protocol Examples

1. Physical layer
   - EIA/TIA-232 · EIA/TIA-449 · ITU-T V-Series · I.430 · I.431 · PDH · SONET/SDH · PON · OTN · DSL · IEEE 802.3 · IEEE 802.11 · IEEE 802.15 · IEEE 802.16 · IEEE 1394 · ITU-T G.hn PHY · USB · Bluetooth · RS-232 · RS-449

2. Data link layer
   - ATM · ARP · IS-IS · SDLC · HDLC · CSLIP · SLIP · GFP · PPI · IEEE 802.2 · LLC · MAC · L2TP · IEEE 802.3 · Frame Relay · ITU-T G.hn DLL · PPP · X.25 LAPB · Q.922 LAPF

3. Network layer
   - IP (IPv4 · IPv6) · ICMP · IPsec · IGMP · IPX · AppleTalk · X.25 PLP

4. Transport layer
   - TCP · UDP · SCTP · DCCP · SPX

5. Session layer
   - Named pipe · NetBIOS · SAP · PPTP · RTP · SOCKS · SPDY

6. Presentation layer
   - MIME · XDR · ASN.1 · ASCII · PGP

7. Application layer
   - NNTP · SIP · SSI · DNS · FTP · Gopher · HTTP · NFS · NTP · SMPP · SMTP · SNMP · Telnet · DHCP · Netconf · more...

---

Slide 27
Von Neumann architecture:

- Messages may contain data and/or instructions.
- The application needs to interpret the messages.
Middleware

- A special layer for distributed systems that covers application, presentation and session layer protocols.
- Offers same APIs to applications
- Hides OS and hardware differences
- Can introduce communication principles other than basic message passing
- Eases the development of distributed software
- Examples:
  - RPC frameworks
  - Distributed databases
  - Actor models
  - Message broker
  - SOAP

Middleware

- Application
- Presentation
- Session
- Transport
- Network
- Data link
- Physical
Middleware can be seen as one system that stretches multiple computers providing the same services on every computer.

Distributed Data Management
Communication

ThorstenPapenbrock
Slide 30
Overview

Communication

- Message Passing
- OSI Model
- **Socket-based Communication**
- Message-oriented Middleware
- Service-oriented Middleware
- Database-oriented Middleware
Socket-based Communication
Low-level Message Passing

Our focus now!
Socket-based Communication
Low-level Message Passing

- Listens on the network.
- Checks every incoming message on whether it is addressed to this node’s MAC.
- If yes, it serves the message to the network layer protocol.
Socket-based Communication
Low-level Message Passing

- Provides system buffer space for messages.
- Checks every incoming message on whether it is addressed to this node’s IP.
- If yes, serves the message to the transport layer protocol.
- If no, finds the IP for the next best hop and serves the message back to the data link protocol.
Socket-based Communication
Low-level Message Passing

- Re-assembles data into logical messages.
- Serves the message to a communication end point provided by the operating system.

- Lots of other services:
  - Connections
  - Reliable message sends (e.g. via re-sending/re-requesting lost messages)
  - Streaming
  - ...

Distributed Data Management
Communication
Thorsten Papenbrock
Slide 35
Socket-based Communication
Low-level Message Passing

- Is triggered when a new message has been copied into a communication end point.
- Splits the message into smaller, equally-sized parts, which better fit low-level buffer sizes (in e.g. the data link layer).
- Serves the message to the network layer.
Socket-based Communication
Low-level Message Passing

- Provides system buffer space for outgoing messages.
- Finds the IP for the next best hop and serves the message to the data link protocol.
- Serves the message to the data link layer.

Distributed Data Management
Communication

Thorsten Papenbrock
Slide 37
- Frames the data with the target’s MAC, a sequence number and a checksum.
- Sends the frame to the physical layer and, hence, the network.
- If messages collide or bits get damaged, the message is resend (sequence numbers and checksums help to detect this).
Socket-based Communication

Sockets

Socket

- A socket is a **communication end point** to which an application can write data that are to be send out over the underlying network, and from which incoming data can be read.
  - Sockets are **buffers** to exchange messages with the transport layer.
  - Sockets are described by **protocol** (usually TCP or UDP), **IP address** and **port number**.
  - Sockets are **symmetric**, i.e., both sender and receiver must speak the same protocol.
  - Think of a file to which the application can hold a handle (sometimes sockets actually have file semantics, as in Java).
  - Both application and transport layer have access to the socket.
    - Read/Write operations to sockets need to be synchronized.
  - Different socket implementations exist, but the interface is standardized.
  - Sockets are by far the most popular form of cross-platform inter-process communication primitives and used within most distributed systems.
  - Date back to RFC 147 (ARPANET, 1971) and Berkeley’s BSD Unix (1983)
Socket-based Communication
Sockets

Sender
Socket
Transport
Network
Data link
Physical

Receiver
Socket
Transport
Network
Data link
Physical

Distributed Data Management
Communication
ThorstenPapenbrock
Slide 40
Socket-based Communication
Sockets and TCP – An Example

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>socket</td>
<td>Create a new connection end point; allocate system resources.</td>
</tr>
<tr>
<td>bind</td>
<td>Associate a specific local address (IP and port) with a socket; if not called, OS will dynamically allocate a port and use the local IP (see client).</td>
</tr>
<tr>
<td>listen</td>
<td>Specify maximum number of pending connections; OS will buffer these requests.</td>
</tr>
<tr>
<td>accept</td>
<td>(blocking) Create a new socket for an arriving request to represent the connection: original socket for further connection requests; new socket for this specific connection (e.g., to fork a dedicated thread).</td>
</tr>
<tr>
<td>connect</td>
<td>(blocking) Attempt to establish a connection with a remote end point; uses a three-way handshake to exchange request, final port number and acknowledgement.</td>
</tr>
<tr>
<td>send</td>
<td>Send some data over the connection.</td>
</tr>
<tr>
<td>receive</td>
<td>Receive some data over the connection.</td>
</tr>
<tr>
<td>close</td>
<td>Release the connection, i.e., socket resources and bindings.</td>
</tr>
</tbody>
</table>

Steps that move a communication to a dedicated socket and, in this way, make it connection-based.
Socket-based Communication

Sockets and TCP – An Example in Python

from socket import *
s = socket(AF_INET, SOCK_STREAM)
s.bind("192.168.0.1", 80)
s.listen(1)
(sc, addr) = s.accept()
while True:
data = sc.recv(1024)
if not data:
    break
sc.send(str(data)+"*")
sc.close()

from socket import *
s = socket(AF_INET, SOCK_STREAM)
s.connect("192.168.0.1", 80)
s.send("hello")
data = conn.recv(1024)
print data
s.close()
Socket-based Communication
Sockets and TCP – An Example in Python

```python
from socket import *
s = socket(AF_INET, SOCK_STREAM)
s.bind(“192.168.0.1”, 80)
s.listen(1)
(sc, addr) = s.accept()
while True:
data = sc.recv(1024)
if not data:
    break
sc.send(str(data)+”*”)
s.close()
```

In reality a bit more complex:
- `send()` and `recv()` calls return when the associated network buffers have been filled (send) or emptied (recv).
  - They do not necessarily handle all the bytes, but tell how many bytes they handled.
  - The application needs to call them again until the entire message has been sent.
- When a `recv` returns 0 bytes, it means the other side has closed (or is in the process of closing) the connection.

➢ One reason to appreciate middleware systems ;-)
Socket-based Communication

Sockets and TCP – An Example in Python

```python
from socket import *
s = socket(AF_INET, SOCK_STREAM)
s.bind("192.168.0.1", 80)
s.listen(1)
(sc, addr) = s.accept()

while True:
data = sc.recv(1024)
if not data:
    break
sc.send(str(data) + "*")
sc.close()
```

```python
from socket import *
s = socket(AF_INET, SOCK_STREAM)
s.connect("192.168.0.1", 80)
s.send("hello")
data = conn.recv(1024)
print data
s.close()
```

```python
# Send
length = len(msg)
s.send(toBytes(length, 8))
byteressent = 0
while byteressent < length:
    sent = s.send(msg[byteressent:])
    byteressent = byteressent + sent

# Receive
length = fromBytes(s.recv(8))
chunks = []
bytesrecv = 0
while bytesrecv < length:
    chunk = s.recv(min(length - bytesrecv, 1048))
    chunks.append(chunk)
    bytesrecv = bytesrecv + len(chunk)
return b''.join(chunks)
```

https://docs.python.org/3/howto/sockets.html
Buffering or No-Buffering?
- Buffering, because a socket pre-allocates buffer space.

Synchronous or Asynchronous?
- Both socket variant exist, but usually synchronous:
  - send() blocks until all data has been written to the socket.
  - receive() blocks until data is available to be read.

Transient or Persistent?
- Transient, because both processes need to be active for messaging.
  - But sockets have a buffer and manage the transmission!
  - Yes, but the message as a whole is not stored and maintained, i.e., if the message is larger than the sockets’ buffers, the transport layer will never see the entire message.
An asynchronous, transient messaging library that supports message queues.

Message queues are variable-length buffers:
- Can take entire messages (unlike TCP, which uses byte streams)
  - Message-awareness
- Enables asynchronicity (sender continues after message submit)
- Despite the asynchronicity, recv()-calls still block if no message is present.

ZeroMQ extends traditional sockets by providing a higher level of abstraction:
- ZeroMQ sockets support many-to-one and one-to-many communication:
  - A socket can be associated with multiple addresses.
  - A socket can connect to multiple addresses/sockets.
- ZeroMQ sockets support pairing of sockets into popular patterns.
Socket-based Communication
ZeroMQ – Communication Patterns

Request-Reply
- A pair of request socket (client) and reply socket (server)
- Every message to the request socket causes a reply to the reply socket.
- Connection handshake is implicit.
- Useful to implement synchronized calls, such as remote procedure calls.

Publish-Subscribe
- A publish socket (server) to which subscribe sockets (client) can subscribe.
- Every message submitted to the publish socket is forwarded to all subscribers.
- Useful to implement multicasting.

Pipeline (or Push-Pull)
- A push socket (server) connected to many pull sockets (clients).
- Every message submitted to the push socket is send to one pull socket; the first client to pull a message receives it.
- Useful to implement task distribution (or collection when used in reverse).
Socket-based Communication
ZeroMQ – Communication Patterns

Request-Reply

- A pair of request socket (client) and reply socket (server)
- Every message to the request socket causes a reply to the reply socket.
- Connection handshake is implicit.

```python
import zmq

context = zmq.Context()
p1 = "tcp://" + HOST + ":" + PORT1  # address 1
p2 = "tcp://" + HOST + ":" + PORT2  # address 2; both can be used to send message to the same socket
s = context.socket(zmq.REP)  # create a reply socket
s.bind(p1)  # bind socket to address 1
s.bind(p2)  # bind socket to address 2

while True:
    message = s.recv()  # block and wait for incoming message; message is the entire message object
    if not "STOP" in message:
        s.send(message + "*")  # interpret message; no need to create a new socket due to message-awareness
    else:
        break
```

Server
Socket-based Communication
ZeroMQ – Communication Patterns

Request-Reply

- A pair of request socket (client) and reply socket (server)
- Every message to the request socket causes a reply to the reply socket.
- Connection handshake is implicit.

```
import zmq
context = zmq.Context()
p1 = "tcp://" + HOST + ":" + PORT1
p2 = "tcp://" + HOST + ":" + PORT2
s = context.socket(zmq.REP)
s.bind(p1)
s.bind(p2)

while True:
    message = s.recv()
    if not "STOP" in message:
        s.send(message + "*")
    else:
        break
```
Overview

Communication

- Message Passing
- OSI Model
- Socket-based Communication
- **Message-oriented Middleware**
- Service-oriented Middleware
- Database-oriented Middleware
Sockets are not suitable for all transport layer protocols (especially not for proprietary protocols used in high-speed networks).

- Transport layer protocols come with different, often complex interfaces (that require e.g. special buffering or synchronization features).
- Transport layer protocols have numerous interfaces:
  - ATP, AppleTalk Transaction Protocol
  - CUDP, Cyclic UDP
  - DCCP, Datagram Congestion Control Protocol
  - FCP, Fibre Channel Protocol
  - IL, IL Protocol
  - MPTCP, Multipath TCP
  - RDP, Reliable Data Protocol
  - RUDP, Reliable User Datagram Protocol
  - SCTP, Stream Control Transmission Protocol
  - SPX, Sequenced Packet Exchange
  - SST, Structured Stream Transport
  - TCP, Transmission Control Protocol
  - UDP, User Datagram Protocol
  - UDP-Lite
  - µTP, Micro Transport Protocol

- Transport layer interfaces are sufficient, but not convenient (every communication involves a lot of redundant code).
- Transport layer buffering capabilities are limited (usually fixed sized buffers where flexible message queues are needed).
Message-oriented Middleware

Middleware

Our focus now!

A special layer for distributed systems that provides a unified API and may offer additional convenience features (w.r.t. buffering, communication protocols, failure handling, ...)

ThorstenPapenbrock
Slide 52
Message-oriented Middleware

- **Message Passing Interface (MPI)**
  - Transient message-passing
  - Focus: high performance

- Actor programming
  - Transient message passing
  - Focus: reactivity, fault-tolerance, maintainability

- Message-queuing systems
  - Persistent message passing
  - Focus: data-intensive, large-scale applications
Message Passing Interface

- A specification for a family of transient message-passing libraries:
  - Popular implementations exist for C, C++ and Fortran
    - e.g. MVAPICH, Open MPI, Intel MPI, IBM Spectrum MPI, ... (most are Linux-based)
  - Most popular middleware for high-performance computing.
- Highly efficient:
  - Supports various transport protocols and their features
  - Can exploit special hardware features
- Highly versatile:
  - Supports buffered/non-buffered communication
  - Supports synchronous/asynchronous communication
- Small weaknesses:
  - Still complex API with many (>440) low-level messaging functions
Message-oriented Middleware

MPI – Core Concepts

Message awareness
- MPI manages messages (of variable length) as entities for transmission, i.e., it sends and receives messages as defined by the application.

Abstract process identifier
- MPI associates a process with an identifier and translates these identifiers transparently into process addresses (i.e., IPs and ports).
- MPI organizes communicating processes in groups, which also receive abstract identifiers.
  - A pair (groupID, processID) uniquely identifies a source/destination.

Transient communication
- MPI does not persist messages and requires all processes to be reachable.
Message-oriented Middleware

MPI – Send and Receive by Example

```c
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[]) {
  int numtasks, rank, dest, source, rc, count, tag=1;
  char inmsg, outmsg='x';
  MPI_Status Stat;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank == 0) {
    dest = 1;
    source = 1;
    rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  } else if (rank == 1) {
    dest = 0;
    source = 0;
    rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
    rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  }

  MPI_Finalize();
}
```

https://computing.llnl.gov/tutorials/mpi/
Many further environment functions exist.

If this is process 0, first send and then receive a message.

If this is process 1, first receive and then send a message.

= all known processes of the cluster
= our process ID in the cluster

```c
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[]) {
    int numtasks, rank, dest, src;
    char inmsg, outmsg='x';
    MPI_Status Stat;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        dest = 1;
        source = 1;
        rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
        rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
    }
    else if (rank == 1) {
        dest = 0;
        source = 0;
        rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
        rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    }
    MPI_Finalize();
}
```
Message-oriented Middleware

**MPI – Synchronization Options**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_isend</td>
<td>Pass a message reference to MPI and continue.</td>
</tr>
<tr>
<td>MPI_bsend</td>
<td>Append message to the local buffer and continue.</td>
</tr>
<tr>
<td>MPI_ssend</td>
<td>Send message and wait until transmission starts.</td>
</tr>
<tr>
<td>MPI_send</td>
<td>Send message and wait until copied to remote buffer.</td>
</tr>
<tr>
<td>MPI_sendrecv</td>
<td>Send message and wait for a reply.</td>
</tr>
</tbody>
</table>

Different MPI libraries may interpret and implement these API semantics differently...
Message-oriented Middleware

MPI – Synchronization Options

Blocking
- Send() call returns if the data was send.
- The message in the send buffer can safely be modified.
- Synchronous
  - The receiving side acknowledged having received the data.
- Asynchronous
  - The system buffer acknowledged having received the data
    (the system buffer copied the data and will make sure it gets send).

Non-Blocking
- Send() call returns immediately.
- The message in the send buffer should not be modified.

In this terminology, RabbitMQ, Kafka, Akka and other JVM-based message broker are non-blocking + asynchronous (messages should not be modified and arrived in the message broker).
Message-oriented Middleware

MPI – Send and Receive by Example

Blocking, synchronous, non-buffered

Blocking, synchronous, buffered

Blocking, asynchronous, buffered

Non-Blocking, asynchronous, buffered
Message-oriented Middleware

Middleware

- Message Passing Interface (MPI)
  - Transient message-passing
  - Focus: high performance

- **Actor programming**
  - Transient message passing
  - Focus: reactivity, fault-tolerance, maintainability

- Message-queuing systems
  - Persistent message passing
  - Focus: data-intensive, large-scale applications
Responsiveness

- The system responds in a timely manner (if possible).
- To user interaction, failures, events, state changes, data characteristics, ...

Resilience

- The system stays responsive in the face of failures.
- Achieved by replication, containment, isolation and delegation.

Elasticity

- The system stays responsive under varying workloads.
- Achieved by dynamic resource (de-)allocation, sharding, replication and decentralization.

Messaging

- The system relies on asynchronous message-passing between loosely coupled, isolated and location transparent components.
- Enables workload distribution, parallelization, failure delegation, load management, and flow control.
Message-oriented Middleware

Actor

Communication
Distributed Data Analytics

Thorsten Papenbrock
Slide 63
Object with strictly private state and behavior

Owns exactly one message queue

Is dynamically scheduled on threads if messages are queued and resources are available

Constitutes the smallest unit of parallelization; execution within an actor is strongly sequential

Reacts to incoming messages; is passive like any object otherwise

Reactions:

- Send a finite number of messages to (other) actors
- Change own state and/or behavior for next message
- Create a finite number of new actors
The actor model is a mathematical, object-oriented message-passing model that treats actors as the universal primitives of concurrent computation.

- Actors in the actor model ...
  - are concurrent, fully encapsulated, self-contained entities.
  - address one another via abstract references that identify an actor object (pointer) inside a process (ID) on some node (IP + port).
- Message-oriented middleware required to i.a. ...
  - resolve abstract addresses.
  - deliver messages from sender to receiver actors.
  - schedule actors on threads.
- Shared memory is strictly forbidden:
  - Actor model helps to prevent many parallel programming issues (concurrent memory access, race conditions, locking, deadlocks, ...)

The actor model follows certain programming principles:

- **Asynchronicity**
  - Actors fire-and-forget messages

- **Encapsulation**
  - Actors have strictly private state and behavior

- **Distribution**
  - Actor locations are transparent

- **Parallelization**
  - Actors execute concurrently

- **Synchronization**
  - Actors synchronize explicitly
Asynchronicity

- Actor-to-Actor communication is asynchronous (fire-and-forget)
- Actor-to-Middleware communication can be synchronous or asynchronous (depending on the actor model implementation)
  - Note: Actor-to-Middleware communication is within the same process and can therefore use shared memory (middleware implementation hides this from developer)
- Middleware communication is usually based on synchronous TCP or UDP sockets
Encapsulation

- Communicating entities have private state and private behavior.
- Communication means sending messages and reacting on received messages.
- Communicate “what” is to be done not “how”!
  - The recipient decides how and if it handles a certain message.
- Communication protocols define the etiquette:
  - Commonly agreed message formats
  - Patterns for message exchanges

It is the developers task to define actor-based application protocols that form viable applications!
### Distribution

- Messages can pass through busses, channels, networks, ...
- Actors use abstract references to identify each other.
- Message-passing system i.e. the middleware resolves addresses and automatically routes messages from senders to receivers.
  - Allows actors to be transparently distributed, i.e., actors do not need to know where their communication partners actually are.
Parallelization

- Actors process one message at a time but different actors operate independently (parallelization between actors not within an actor).
- Actors may spawn new actors if needed (dynamic parallelization).
- **Task parallelism:**
  - Different actors execute different tasks.
- **Data parallelism:**
  - Different actors execute the same task on different data elements.
Synchronization

- Synchronization happens explicitly via messaging and state changes.
- Immutable messages and private actor states prevent concurrency conflicts (e.g. concurrent memory access).
- Deadlocks, starvation and live-locks are easier to avoid with message passing, because resource ownership and waiting behavior is more explicit; still all are possible if communication protocols are faulty implemented.
- Actor-Middleware and Middleware-Actor communication still requires synchronization primitives or lock-free data structures.
  - Message passing can be implemented via locking/atomic operations on the message queue(s) or buffer(s).
  - Vice versa: Locks can be implemented via exchange of messages (e.g. the request/response pattern introduces waiting behavior).

Message-oriented Middleware
Actor Model Principles

Actor
(Thread)

Middleware
(Thread)

```
ReentrantLock messageMutex = new ReentrantLock();
Queue<Work> messageQueue = new ArrayDeque<Work>();
```

```
try {
  messageMutex.lock();
  messageQueue.offer(message);
} finally {
  messageMutex.unlock();
}
```

```
try {
  messageMutex.lock();
  message = messageQueue.poll();
} finally {
  messageMutex.unlock();
}
```

Middleware
(Distributed Data Management
Communication

Arbitrary synchronization primitive (mutex, semaphore, barrier, monitor, ...)

= Shared Memory

= Critical Section

ReentrantLock messageMutex
Queue<Work> messageQueue

ReentrantLock messageMutex = new ReentrantLock();
Queue<Work> messageQueue = new ArrayDeque<Work>();

try {
  messageMutex.lock();
  messageQueue.offer(message);
} finally {
  messageMutex.unlock();
}

try {
  messageMutex.lock();
  message = messageQueue.poll();
} finally {
  messageMutex.unlock();
}
Message-oriented Middleware
Actor Model Principles

Actor
(Thread)

Middleware
(Thread)

private void send(Message message) {
    myQueue.add(message);
}

private Message receive() {
    while (true) {
        message = myQueue.remove();
        if (message != null)
            return message;
    }

BlockingQueue<Message> myQueue = new BlockingQueue<Message>;

Blocks if middleware reads from queue.

Blocks if actor reads from queue.

Lock-based Queue

Distributed Data Management
Communication

ThorstenPapenbrock
Slide 73
Message-oriented Middleware
Actor Model Principles

Actor (Thread)
NonBlockingQueue<Message> myQueue = new NonBlockingQueue<Message>;

private void send(Message message) {
    myQueue.add(message);
}

Middleware (Thread)
private Message receive() {
    while (true) {
        message = myQueue.remove();
        if (message != null)
            return message;
    }

NonBlockingQueue
- A non-blocking Queue data structure
- Based on the atomic compare-and-set operation
- Extension of the TreiberStack* data structure

public class NonBlockingQueue<T> {
    private final AtomicReference<Node<T>> head;
    private final AtomicReference<Node<T>> tail;

    public NonBlockingQueue() {
        this.head = new AtomicReference<T>(null);
        this.tail = new AtomicReference<T>(null);
    }

    private class Node<T> {
        public volatile T message;
        public volatile Node<T> next;
        public volatile Node<T> previous;

        public Node(T message) {
            this.message = message;
            this.next = null;
        }
    }

    public void add(T message) { … }
    public T remove() { … }
}

public void add(T message) { … }
Node<T> newNode = new Node<T>(message);
Node<T> currentTail;
    do {
        currentTail = this.tail.get();
        node.previous = currentTail;
    } while (!this.tail.compareAndSet(currentTail, newNode));

    if (newNode.previous != null)
        newNode.previous.next = newNode;

    this.head.compareAndSet(null, newNode);
}

public T remove() { … }
    if (this.head.get() == null)
        return null;

    Node<T> currentHead, nextNode;
    do {
        currentHead = this.head.get();
        nextNode = currentHead.next;
    } while (!this.head.compareAndSet(currentHead, nextNode));

    return currentHead.getValue();
}
Message-oriented Middleware

Actor Model Implementations

- **Erlang:**
  - Actor library natively included in the Erlang language
  - First popular actor implementation
  - Special: Native language support and strong actor isolation

- **Akka:**
  - Actor library for the JVM (Java and Scala)
  - Most popular actor implementation (a.t.m.)
  - Special: Actor hierarchies and typed actors

- **Orleans:**
  - Actor library for Microsoft .NET/C#
  - Very popular in research and industry (due to Microsoft)
  - Special: Virtual actors (persistent state and transparent location)
Message-oriented Middleware

Actor Model Implementations

- **Erlang:**
  - Actor library natively included in the Erlang language
  - First popular actor implementation
  - Special: **Native language support** and strong actor isolation

- **Akka:**
  - Actor library for the JVM (Java und Scala)
  - Most popular actor implementation (a.t.m.)
  - Special: **Actor hierarchies** and typed actors

- **Orleans:**
  - Actor library for Microsoft .NET/C#
  - Very popular in research and industry (due to Microsoft)
  - Special: **Virtual actors** (persistent state and transparent location)
Message-oriented Middleware
Actor Model – Akka

- Implements the actor modell
- A free and open-source toolkit and runtime for building concurrent and distributed applications on the JVM ([https://akka.io/](https://akka.io/))
- Written in Scala ([https://scala-lang.org/](https://scala-lang.org/))
- Offers APIs for Java und Scala
- Invented by Jonas Bonér
- Maintained by Lightbend ([https://lightbend.com/](https://lightbend.com/))
message-oriented Middleware
Actor Model – Akka

```
public class Worker extends AbstractActor {
    @Override
    public Receive createReceive() {
        return receiveBuilder()
            .match(String.class, this::respondTo)
            .matchAny(object -> System.out.println("Could not understand received message"))
            .build();
    }
    private void respondTo(String message) {
        System.out.println(message);
        this.sender().tell("Received your message, thank you!", this.self());
    }
}
```

Inherit default actor behavior, state and mailbox implementation

The Receive class performs pattern matching and de-serialization

A builder pattern for constructing a Receive object with otherwise many constructor arguments

Send a response to the sender of the last message (asynchronously, non-blocking)
public class Worker extends AbstractActor {
    @Override
    public Receive createReceive() {
        return receiveBuilder()
            .match(String.class, s -> this.sender().tell("Hello!", this.self()))
            .match(Integer.class, i -> this.sender().tell(i * i, this.self()))
            .match(Double.class, d -> this.sender().tell(d > 0 ? d : 0, this.self()))
            .match(MyMessage.class, s -> this.sender().tell(new YourMessage(), this.self()))
            .matchAny(object -> System.out.println("Could not understand received message"))
            .build();
    }
}
public class Worker extends AbstractActor {

    public static class MyMessage implements Serializable {} 

    @Override
    public Receive createReceive() {
        return receiveBuilder()
            .match(MyMessage.class, s -> this.sender().tell(new OtherActor.YourMessage(), this.self()))
            .matchAny(object -> System.out.println("Could not understand received message"))
            .build();
    }
}
Message-oriented Middleware
Actor Model – Akka

Example: A flight booking system
Message-oriented Middleware
Actor Model – Akka

Example: A flight booking system

- Booking API
  - Booking Request
  - Booking Task
  - Booking Worker
  - Booking Data Parallelism

- Email
- Banking API
- Reservations
- Promotion
- Promotion Actor
- Billing Actor
- Payment Actor
- Reservation Actor
- Task Parallelism

- Bill
- Pay
- Reserve
- Promote
- Distributed Data Management
- Communication

ThorstenPapenbrock
Slide 83
ActorSystem

Dynamically schedules actors with messages on threads (transparent multi-threading):
- \#threads \approx \#CPU cores
- \#actors > \#CPU cores (usually many hundreds)
  ➢ over-provisioning!

User actors live here

System and remote actors live here

Parents supervise their children

Booking Processor
Billing Actor
Payment Actor
Reservation Actor
Promotion Actor

Booking Worker

Dispatcher

Thread 1
Thread 2
Thread 3
Thread 4
Reactive Programming

- A declarative programming paradigm based on event and message flows.
- Program components (= actors) declare how they react on certain messages.
- Instead of following a fixed calculation plan, components behave dynamically and independent of each other.
  - Writing a reactive algorithm is more like declaring rules for how to react on certain input changes rather than defining a step-by-step execution plan.
- Reactions are triggered by data or changes in the environment.
- Reaction:
  - changing state (= private variables)
  - changing behavior (= private code)
  - changing algorithm topology (= child actors)
  - sending further messages
- Reactivity helps to optimize resource utilization, robustness and elasticity.
"Let it crash" philosophy

- Distributed systems are inherently prone to errors (because there is simply more to go wrong/break).
  - Message loss, unreachable mailboxes, crashing actors ...
- Make sure that critical code is supervised by some entity that knows how errors can be handled.
- Then, if an error occurs, do not (desperately) try to fix it: let it crash!
  - Errors are propagated to supervisors that can deal better with them.
- Example: Actor discovers a parsing error and crashes.
  - Maybe message was corrupted in message transfer.
  - Its supervisor restarts the actor and resends the message.

Fault tolerance tools

- Lifecycle management (e.g. automatic restart of failed actors)
- Supervision (let it crash)
- Dead letters (e.g. resent/re-route of failed messages)
Akka hands-on:

- Demo
- Actor lifecycle
- Messaging guarantees
- Fault tolerance
- Remoting
- Scheduling
- Patterns
  - Maszer/Worker
  - Reaper
  - Proxy
  - Singleton
  - ...
Message-oriented Middleware

Middleware

- Message Passing Interface (MPI)
  - Transient message-passing
  - Focus: high performance

- Actor programming
  - Transient message passing
  - Focus: reactivity, fault-tolerance, maintainability

- Message-queuing systems
  - Persistent message passing
  - Focus: data-intensive, large-scale applications
A **message-queuing system** is a message-oriented middleware that provides various services for (persistent) asynchronous communication.

- **Message queue:**
  - Intermediate-term storage capacity maintained by the middleware
  - Stores messages until delivered to (all) recipients
  - Sometimes tied to sender and/or receiver (see actor model); sometimes subscription-based (see RabbitMQ)
  - Holds a logical, location-independent, system-wide unique identifier

- **Queue manager:**
  - Part of the message-queuing system that handles queue lifecycles and all message traffic
  - A separate process and/or library that is linked into the application
  - Message-queuing system requires a queue manager on every node
A message-queuing system is a message-oriented middleware that provides various services for asynchronous communication. This includes:

- **Message queue:**
  - Intermediate-term storage capacity maintained by the middleware
  - Stores messages until delivered to (all) recipients
  - Sometimes tied to sender and/or receiver (see actor model); sometimes subscription-based (see RabbitMQ)
  - Holds a logical, location-independent, system-wide unique identifier

- **Queue manager:**
  - Part of the message-queuing system that handles queue lifecycles and all message traffic
  - A separate process and/or library that is linked into the application
  - Message-queuing system requires a queue manager on every node

**Strictly speaking:**

If all queue managers run within application processes, i.e., there is no separate queue manager process, then the message transfer requires both sender and receiver process to run simultaneously (at some point in time).

- Can be considered non-persistent

(although local queue manager can in principle wait with the transfer until the target system is up without blocking local application threads)
A message-queuing system is a message-oriented middleware that provides various services for (persistent) asynchronous communication.

- **Sender/receiver:**
  - May not be active at the same time
  - May be programmed in different languages (in some MQ systems)

- **Messages:**
  - Need to be understood by sender/receiver
  - May not be understood by the message-queuing system (byte arrays)
  - Need to be properly addressed to a message queue (unique ID)

- **Messaging:**
  - Message transfers may take longer than with Sockets, MPI, RPCs, ...
  - Middleware guarantees that the message will eventually be delivered to the recipient (no guarantee on when or if the message is read).
A message-queuing system is a message-oriented middleware that provides various services for (persistent) asynchronous communication.

- **(Basic) Interface:**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>put</td>
<td>Append a message to a specific queue (non-blocking).</td>
</tr>
<tr>
<td>get</td>
<td>Remove first message from a queue (blocking)</td>
</tr>
<tr>
<td>poll</td>
<td>Try to remove first message from a queue (non-blocking).</td>
</tr>
<tr>
<td>notify</td>
<td>Install a handler that is called when a message is put into a queue.</td>
</tr>
</tbody>
</table>

- **Variations:**
  - get/poll may take but not remove the element
  - get/poll may not return the first but a different element (based on priority, a search pattern, sender or index)
Message-oriented Middleware
Message-Queuing Systems – Tasks

- **Message Buffering:**
  - Maintenance of message queues and their subscribers/owners

- **Message Delivery:**
  - Answering of poll-requests or notification of recipients
  - (sometimes) Ensuring reliable message sends (message loss detection and resending)
  - (sometimes) Enabling synchronous message sends

- **Address Resolution:**
  - Translation of symbolic addresses into physical references, ports and IP-addresses (enables location transparent communication)

- **Data Transmission:**
  - Routing: one-to-one messages
  - Broadcasting: one-to-many messages

- **Encoding:**
  - Serialization and deserialization for messages that are send across process boundaries
- A message queue lives in some queue manager.
- Applications can write/read only local queues.
  - To write/read remote queues, queue manager create local queues (usually dynamically and transparently).
  - In the example: Sender sends to a remote target queue.
- Each pair of communicating processes maintains (usually) exactly one transport-level connection (e.g. TCP) for all traffic between arbitrary queues.
- A process maintains (usually) exactly one send queue for each connection.
- Address lookup database (or routing table):
  - stores a mapping of logical queue IDs to physical addresses (e.g. transport protocol, host and port).
  - can be a replicated store or a network service.
Message-oriented Middleware
Message-Queuing Systems – Transmission

- **Routers:**
  - are queue managers that forward messages addressed to non-local message queues.
  - are needed if the address lookup database is incomplete or network topologies are complex so that messages cannot be sent directly to the target queues.
Queue manager in different processes and message passing between application and middleware
Queue manager in different processes and RPC between application and middleware
Distributed systems usually consist of more than two nodes!

- Assume we want to integrate different, potentially heterogeneous systems into a single, coherent distributed information system.
- Each system speaks its own protocols and message formats.
- Due to the complexity and abstractions made by middleware layers, it is often not possible to let all systems agree on one communication protocol.
Solution 1: Wrappers

- Peer-to-peer communication channels between all systems using wrappers
- Requires $O(N^2)$ wrappers
- Works well for homogeneous systems (like MPI or actor systems) but causes scalability issues and code duplication for heterogeneous systems

Solution 2: Message Broker

- Centralized communication hub with message translation capabilities
- Requires $O(N)$ wrappers
- Is slower than peer-to-peer communication
Message Broker

- An application-level gateway on top of a message-queuing system that routes and translates messages.
- The message-queuing system treats the broker as yet another application.
- Message-queuing system plus message broker are one form of message-oriented middleware:

![Diagram of Message-oriented Middleware and related components]
Message Broker (cont.)

- Can add additional capabilities on top of the message-queuing system (often via plugins and rule sets), such as:
  - Message transformations/re-encodings
  - Replication of queues
  - Partitioning of queues
  - Sender and receiver group management
  - Batch processing
- Example: Apache Kafka
Message-oriented Middleware
Message-Queueing Systems – Message Broker

Sender Process

Sender Application

Interface

Queue Manager

Queue Manager

OS

Broker Process

Broker Application

Plugins

Rules

Interface

Queue Manager

Receiver Process

Receiver Application

Interface

Queue Manager

OS

Thorsten Papenbrock
Slide 102
Publish-Subscribe

- The queue manager of the message broker maintains all message queues.
  - Any process can potentially access all queues of the broker.
  - Queues may be used in many-to-many communications.
- Publish-subscribe is a message-queuing communication pattern:
  - Senders (called publishers) address their messages to queues, which represent topics or categories, without knowing the real recipients.
  - Receivers (called subscribers) listen on one or multiple queues via subscriptions to these queues; they consume messages without knowing the sender(s).
- Subscription = registered callback function
General message delivery

- Processes can ...
  - create named message queues.
  - subscribe to existing message queues.
  - send messages to a queue.
- The message broker promises that send messages are delivered to some (1-to-1) or all (broadcasting) subscribers of a queue.

Popular message brokers

- Commercial:
  - TIBCO, IBM WebSphere, webMethods, ...
- Open source:
  - Apache Kafka, RabbitMQ, ActiveMQ, HornetQ, NATS, ...
Message-oriented Middleware

Example: RabbitMQ – Sending a Message

```java
public class Send {

    private final static String QUEUE_NAME = "hello";

    public static void main(String[] argv) throws Exception {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("localhost");
        Connection connection = factory.newConnection();
        Channel channel = connection.createChannel();

        channel.queueDeclare(QUEUE_NAME, false, false, false, null);
        String message = "Hello World!";
        channel.basicPublish("", QUEUE_NAME, null, message.getBytes("UTF-8"));
        System.out.println("[x] Sent " + message + "]");

        channel.close();
        connection.close();
    }
}
```

Create a **connection** to the message broker running on localhost (see TCP protocol).

Create a **channel** to a queue; the queue is created if it does not exist yet.

Send the message encoded as an array of bytes.

Close all channels and the connection.

https://www.rabbitmq.com/getstarted.html
Message-oriented Middleware

Example: RabbitMQ – Receiving a Message

```java
public classRecv{

private final static String QUEUE_NAME = "hello";

public static voidmain(String[] argv) throws Exception {
    ConnectionFactory factory = new ConnectionFactory();
    factory.setHost("localhost");
    Connection connection = factory.newConnection();
    Channel channel = connection.createChannel();
    channel.queueDeclare(QUEUE_NAME, false, false, false, null);
    System.out.println("[*] Waiting for messages. To exit press CTRL+C");

    Consumer consumer = new DefaultConsumer(channel) {
        @Override
        public voidhandleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, byte[] body)
        throws IOException {
            String message = new String(body, "UTF-8");
            System.out.println("[X] Received "+ message + ";");
        }
    };
    channel.basicConsume(QUEUE_NAME, true, consumer);
}
```

Create a connection to the message broker running on localhost (see TCP protocol).
Create a channel to a queue; the queue is created if it does not exist yet.
Create a callback object that can buffer and consume messages from a queue.
Special metadata for the received message:
- E.g. encoding, timestamp, sender, priority, ...
Decode and print any received byte message.
Subscribe the new consumer to the queue; the broker will call it with messages of that queue.

https://www.rabbitmq.com/getstarted.html
Message-oriented Middleware

Example: RabbitMQ – Example in Python

```python
#!/usr/bin/env python
import pika

connection = pika.BlockingConnection(pika.ConnectionParameters(
    host='localhost'))
channel = connection.channel()

channel.queue_declare(queue='hello')
channel.basic_publish(exchange='',
    routing_key='hello',
    body='Hello World!'
)
print(" [x] Sent 'Hello World!'")
connection.close()
```

Further APIs:
Ruby, PHP, C#, JavaScript, Go, Elixir, Objective-C, Swift, …

https://www.rabbitmq.com/getstarted.html
Message-oriented Middleware
Message-Queuing Systems – Message Broker

Advantages
- **Maintainability:** Decouples sender and receiver objects/threads/processes
- **Asynchronicity:** Buffers messages if receiver is unavailable or overloaded
- **Robustness:** May redirect messages if some receiver is unreachable
- **Efficiency:** Message routing, buffering, replication... can be optimized

Disadvantages
- **Scalability:** Message broker and shared queues can be bottlenecks
- **Latency:** Message broker is an additional hop for each message
- **Reliability:** Publishers cannot be sure that their messages will be consumed; broker will need to drop undeliverable messages eventually
Overview
Communication

- Message Passing
- OSI Model
- Socket-based Communication
- Message-oriented Middleware
- **Service-oriented Middleware**
- Database-oriented Middleware
Service

- A well-defined API that can be accessed by other (remote) processes
- Identified by service protocol + IP + port
- Offers functions that may take arguments (= a send message) and return values (= a receive message)
  - Functions define fine-grained restrictions on what can be communicated.
  - Functions imply clear actions (whereas messages that imply facts).
  - Functions are blocking and synchronous

Asymmetric Communication

- Communicating processes have two roles:
  - **Server**: exposes a service that other processes can see and use.
  - **Client**: connects to a server’s service and calls functions.
Client knows:
- How to address the server (IP + Port)
- How to send data (serialization + packaging)

Client does not (yet) know:
- What functions are available
- What data it needs to send to call a function
Client does not (yet) know:

- What functions are available
- What data it needs to send to call a function
Service-oriented Middleware
Service-oriented Communication

- Client does not (yet) know:
  - What functions are available
  - What data it needs to send to call a function

Protocol:
- function call $\rightarrow$ data
- data $\rightarrow$ function call (w.r.t. given interface)
A protocol that allows processes to directly call functions in remote processes (i.e., cause procedures to execute in different address spaces).

- The object-oriented equivalent is remote method invocation (RMI).
- Remote procedures are called like normal (local) procedures.
  - Tight coupling between processes

Bruce Jay Nelson introduced the RPC idea in 1984.
The RPC middleware:
- requires the service’s interface on server and client.
- implements the protocol for transmitting a function call.
- uses the interface to automatically generate two proxies:
  - **Stub** (function call → data)
    - Implements and offers the interface functions.
    - Translates any function call into a message and sends it to the skeleton.
      - function/parameter marshaling
  - **Skeleton** (data → function call)
    - Implements a messaging-based endpoint for a service.
    - Translates any message into a function call and maps it to the right local function implementation.
1. **Client** calls a remote procedure and waits.
2. **Stub** accepts the procedure call and serializes both the call and its parameters.
3. **RPC Runtime** sends the serialized call via TCP/UDP to the server.
4. **Skeleton** accepts procedure call, deserializes the message and calls the corresponding service procedure with the given parameters.
5. **Server** handles the call and returns a result.
6. **Skeleton** accepts the result and serializes it.
7. **RPC Runtime** sends the serialized result via TCP/UDP back to the client.
8. **Stub** accepts the result, deserializes it and forwards it to the waiting client.
9. **Client** awakes and accepts the result.
1. **Client** calls a remote procedure and waits.
2. **Stub** accepts the procedure call and serializes both the call and its parameters.
3. **RPC Runtime** sends the serialized call via TCP/UDP to the server.
4. **Skeleton** accepts the procedure call, deserializes the message and calls the corresponding service procedure with the given parameters.
5. **Server** handles the call and returns a result.
6. **Skeleton** accepts the result and serializes it.
7. **RPC Runtime** sends the serialized result via TCP/UDP back to the client.
8. **Stub** accepts the result, deserializes it and forwards it to the waiting client.
9. **Client** awakes and accepts the result.

**Service-oriented Middleware**

**Remote Procedure Call (RPC)**

- **Client code**
- **Server code**
- **Service**
- **RPC calls block the client**

**Thorsten Pappenbrock**

Slide 117
Rendezvous protocol

- Handshake protocol for sending large blocks of data via synchronous communication.
- Avoid sending data to processes that cannot accept them (at the moment).
- Before data is sent, the receiver needs to acknowledge that it is ready to accept data.
RPC/RMI are protocols of which many framework implementations exist.

RPC/RMI provide a communication interface in the programming language and hide the communication protocol in a runtime, i.e., middleware.

RPC/RMI implementations can be:
- language specific: interface is written in same language; often the programming language itself
- language agnostic: interface is written in some RPC/RMI dialect; compiles to different programming languages

The RPC/RMI protocols use blocking, synchronous communication but it is easy to turn the idea into non-blocking, asynchronous communication:
- e.g. procedure calls may immediately return “Future” or “Promise” objects
Strengths of RPC/RMI

- RPC/RMI frameworks are well suited for machine to machine communication (remote calls appear like local calls; program does not leave its own language).

- RPC/RMI frameworks are easy to use (automatic code generation and abstraction of the messaging details).

- RPC/RMI frameworks are extensive (no restrictions other than those the interface language has).

- RPC/RMI frameworks offer good performance (highly optimized messaging, because the runtime controls both ends of the communication and no third party needs to understand the messages).
Weaknesses of RPC/RMI

- RPC/RMI cause a **tight coupling of server and client code.** (interface changes always concern both)

- **Local and remote function calls are, in fact, very different.**
  - Local function calls are predictable: they succeed or fail, throw proper exceptions or starve processing; can handle same pointers and data types than caller
  - Remote function calls are unpredictable: they fail silently, succeed but responses get lost, are unavailable; cannot handle the caller’s pointers (and all data types)

- RPC/RMI code may be **hard to debug and test.** (code generation; possibly hiding of network errors)

Good/modern RPC frameworks make differences explicit and forward errors transparently so that application code can (and should!) handle these issues.
## Service-oriented Middleware

### Remote Procedure Call (RPC)

<table>
<thead>
<tr>
<th>Language-specific</th>
<th>[edit]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java's Java Remote Method Invocation (Java RMI) API provides similar functionality to standard Unix RPC methods.</td>
<td></td>
</tr>
<tr>
<td>Modula-3's network objects, which were the basis for Java's RMI[^10]</td>
<td></td>
</tr>
<tr>
<td>RPyC implements RPC mechanisms in Python, with support for asynchronous calls.</td>
<td></td>
</tr>
<tr>
<td>Distributed Ruby (DRb) allows Ruby programs to communicate with each other on the same machine or over a network. DRb uses remote method invocation (RMI) to pass commands and data between processes.</td>
<td></td>
</tr>
<tr>
<td>Erlang is process oriented and natively supports distribution and RPCs via message passing between nodes and local processes alike.</td>
<td></td>
</tr>
<tr>
<td>Elixir builds on top of the Erlang VM and allows process communication (Elixir/Erlang processes, not OS processes) of the same network out-of-the-box via Agents and message passing.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Application-specific</th>
<th>[edit]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action Message Format (AMF) allows Adobe Flex applications to communicate with back-ends or other applications that support AMF.</td>
<td></td>
</tr>
<tr>
<td>Remote Function Call is the standard SAP Interface for communication between SAP systems. RFC calls a function to be executed in a remote system.</td>
<td></td>
</tr>
</tbody>
</table>

### General | [edit] |
| NFS (Network File System) is one of the most prominent users of RPC |
| Open Network Computing Remote Procedure Call, by Sun Microsystems |
| D-Bus open source IPC program provides similar function to CORBA |
| SORCER provides the API and execution-oriented language (EOL) for a federated method invocation |
| XML-RPC is an RPC protocol that uses XML to encode its calls and HTTP as a transport mechanism. |
| JSON-RPC is an RPC protocol that uses JSON-encoded messages |
| JSON-RPC is an RPC protocol that uses JSON-encoded messages |
| SOAP is a successor of XML-RPC and also uses XML to encode its HTTP-based calls. |
| ZeroC's Internet Communications Engine (Ice) distributed computing platform. |
| Etch framework for building network services. |
| Apache Thrift protocol and framework. |
| CORBA provides remote procedure invocation through an intermediate layer called the object request broker. |
| Libevent provides a framework for creating RPC servers and clients.[^11] |
| Windows Communication Foundation is an application programming interface in the .NET framework for building connected, service-oriented applications. |
| Microsoft .NET Remoting offers RPC facilities for distributed systems implemented on the Windows platform. It has been superseded by WCF. |
| The Microsoft DCOM uses MSRPC which is based on DCE/RPC |
| The Open Software Foundation DCE/RPC Distributed Computing Environment (also implemented by Microsoft). |
| Google Protocol Buffers (protobufs) package includes an interface definition language used for its RPC protocols[^12] open sourced in 2015 as gRPC[^13] |
| Google Web Toolkit uses an asynchronous RPC to communicate to the server service. |
| Apache Auro provides RPC where client and server exchange schemas in the connection handshake and code generation is not required. |
| Embedded RPC is lightweight RPC implementation developed by NXP, targeting primary CortexM cores |

[^10]: Java's Remote Method Invocation (RMI) API is a standard that allows Java objects to be invoked remotely across a network. It is part of the Java Development Kit (JDK) and is used to create distributed applications. The RMI API provides a framework for creating distributed objects and services, enabling the creation of remote clients and servers. The server exposes methods that can be called by remote clients, and the client can invoke these methods as if they were local to the client. RMI is based on the Java RMI package and is used for creating distributed applications that require remote method calls.

[^11]: Libevent is a library for building event-driven servers and other applications that require a concurrent event loop. It is designed to abstract the underlying event loop model from event handlers, allowing different event loop models (such as BSD sockets, epoll, kqueue, or poll) to be used interchangeably.

[^12]: Google Protocol Buffers (protobufs) is a software library and runtime framework for serializing structured data. It is used to transmit data between processes or systems, and it can be used with any language that supports serialization. Protocol Buffers is designed to be small and fast, and it is able to serialize data efficiently, making it suitable for use in distributed systems and other environments where performance is critical.

[^13]: gRPC (Google Remote Procedure Call) is a high-performance, open-source RPC framework developed by Google. It is designed to be used with gRPC libraries in C++, .NET, Java, Python, Ruby, and Go, and it can be used with any language that supports RPC. gRPC is designed to be fast and efficient, and it uses a binary protocol to serialize and deserialize data, making it suitable for use in high-performance applications.

RPC implementations

- Thrift-based
- Protocol Buffers-based
- Avro-based
Service-oriented Middleware

Service-Oriented Architecture (SOA)

- A server process can, again, become a client to some other server.
  - (Distributed) systems of interacting processes
- Services should be self-contained black box components that represent logical activities hiding lower-level services.
- Microservice architecture:
  - Variant of SOA where services are particularly fine-grained and the protocol is lightweight

Examples

Web Browser

Apps

Online Games

Distributed Data Management
Communication

ThorstenPapenbrock
Slide 123
Microservice architecture

- Example

A light-weight application framework for Java with support for microservice development

https://piotrminkowski.wordpress.com/2017/05/29/spring-cloud-microservices-at-pivotal-platform/
Microservice architecture

- Example: *Mastering Chaos - A Netflix Guide to Microservices*

https://www.youtube.com/watch?v=CZ3wIuvmHeM
(Micro-)Services are:

- loosely coupled
- independently deployable
- heterogeneous in their implementation (languages, libraries, resources, dependencies, ...)
- dependent on only lightweight protocols
- highly maintainable and testable
- organized around business capabilities
- often owned by a small team

(Micro-)Services enable:

- rapid, frequent and reliable delivery of large, complex applications
- the evolution of an applications technology stack (at runtime).
HTTP
- Used by the largest SOA systems on the planet, e.g., the World Wide Web.

(HTTP) REST
- If you need clearer conventions for HTTP service APIs.
  (e.g. to make them easier to maintain and better machine consumable)
- Used by many Web applications to connect front- and backend systems.

(HTTP + RPC) SOAP
- If you develop heterogeneous distributed systems that need to communicate not only data but also instructions (i.e., method calls).
- Used by many large scale, heterogeneous distributed systems.
Service-oriented Middleware

Hypertext Transfer Protocol (HTTP)

Definition

- A stateless, synchronous request-response application protocol for distributed, collaborative, and hypermedia information systems
- The foundation for communication in the World Wide Web
- Hypertext: structured text that uses logical links (hyperlinks) between nodes containing text (usually HTML)

Technical Details

- Message format: designed for hypertext, but works for any text format
- Based on the TCP transport layer protocol
- Uniform Resource Locators (URLs) / Uniform Resource Identifier (URI) to find services and resources:
  - scheme://[user[:password]@]host[:port]/path
  - E.g.: http://hpi.de/naumann/people/thorsten-papenbrock
HTTP defines the message format and protocol so that two different HTTP implementations, i.e., different runtimes can communicate.

Library/program that implements the well defined HTTP interface (e.g. web browser, curl, libashttp, java.net.HttpURLConnection)

Well defined functions: GET, POST, PUT, DELETE, ...

Well defined messages: header fields and text data
HTTPs

- HTTP over Transport Layer Security (TLS) / Secure Sockets Layer (SSL)
- Features:
  - Privacy through symmetric encryption
  - Authentication through public-key cryptography
  - Integrity through checking of message authentication codes

Session (HTTP/1.1)

- A sequence of network request-response transactions:
  1. Client establishes a TCP connection to server port (typically port 80).
  2. Client sends an HTTP message.
  3. Server sends back a status line with a message of its own.
  4. Client sends next HTTP message or closes the TCP connection.
Request Message Pattern

- A request-line: `<method> <resource identifier> <protocol version>`
- Any header lines: `<header field>: <value>`
- An empty line
- A message-body: `<any text format>`

Request Methods

- **GET**: Retrieve information from the target resource using a given URI (no side effects).
- **HEAD**: Like GET, but response contains only status line and header section (no content).
- **POST**: Send data to the target resource; the resource decides what to do with the data.
- **PUT**: Send data to the target resource; replace the content of the resource with that data.
- **DELETE**: Removes all content of the target resource.
- **CONNECT**: Establishes a tunnel to the server identified by a given URI.
- **OPTIONS**: Describe the communication options for the target resource.
- **TRACE**: Performs a message loop back test along with the path to the target resource.
Service-oriented Middleware
Hypertext Transfer Protocol (HTTP)

Request Message Pattern

- A request-line:
  `<method> <resource identifier> <protocol version>`
- Any header lines:
  `<header field>: <value>`
- An empty line
- A message-body:
  `<any text format>`  
  optional

Examples

- GET http://hpi.de/naumann/people/thorsten-papenbrock/publications HTTP/1.1
  → absolute URI: for requests to a proxy, which should forward the request
  → no additional header fields

- GET /naumann/people/thorsten-papenbrock/publications HTTP/1.1
  User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
  Host: www.hpi.de:80
  Accept-Language: en-us
  → relative URI: for request to origin server
  → some header fields as example
Request Message Pattern

- A request-line: `<method> <resource identifier> <protocol version>`
- Any header lines: `<header field>: <value>`
- An empty line
- A message-body: `<any text format>`

Examples

- POST /naumann/people/thorsten-papenbrock/publications HTTP/1.1
  Host: www.hpi.de:80
  Content-Type: text/xml; charset=utf-8
  Accept-Language: en-us
  Accept-Encoding: gzip, deflate
  Connection: Keep-Alive

  <publication>A Hybrid Approach to Functional Dependency Discovery</publication>

  → post a new publication entry to the publications resource (should be appended)
  → flags indicate utf-8 formatted xml content and ask to keep the connection open

PUT would replace all publications with the new one
Response Message Pattern

- A status-line: `<protocol version> <status code> <reason-phrase>`
- Any header lines: `<header field>: <value>`
- An empty line
- A message-body: `<any text format>` optional

Status codes

- 1xx: Informational: the request was received and the process is continuing.
- 2xx: Success: the action was successfully received, understood, and accepted.
- 3xx: Redirection: further action must be taken in order to complete the request.
- 4xx: Client Error: the request contains incorrect syntax or cannot be fulfilled.
- 5xx: Server Error: the server failed to fulfill an apparently valid request.
Response Message Pattern

- A status-line: `<protocol version> <status code> <reason-phrase>`
- Any header line: `<header field>: <value>`
- An empty line
- A message-body: `<any text format>`

Example


```
HTTP/1.1 200 OK
Date: Mon, 24 Jul 2017 12:28:53 GMT
Server: Apache/2.2.14 (Win32)
Last-Modified: Sat, 22 Jul 2017 13:15:56 GMT
Content-Length: 98
Content-Type: text/html
Connection: Closed

<html><body><h1>Welcome to my homepage!</h1></body></html>
```
The cURL Program

- Library and command-line tool for transferring data using various protocols
- Originally developed as “see url” in 1997
- Examples:
  - `curl -i -X GET http://localhost:8080/datasets`
  - `curl -i -X GET http://localhost:8080/datasets/by/csv`
  - `curl -i -X POST -d '{"name":"Planets","ending":"csv","path":"datasets"}'
    -H 'Content-Type:application/json; charset=UTF-8'
    http://localhost:8080/datasets`
  - `curl -i -X DELETE http://localhost:8080/datasets/1`
  - `curl -i -X GET http://localhost:8080/datasets/1`
  - `curl -i -X PUT -d '{"name":"Planets","ending":"csv","path":"datasets"}'
    -H 'Content-Type:application/json; charset=UTF-8'
    http://localhost:8080/datasets/1`
Service-oriented Middleware
Popular Service Protocols

HTTP
- Used by the largest SOA systems on the planet, e.g., the World Wide Web.

(HTTP) REST
- If you need clearer conventions for HTTP service APIs.
  (e.g. to make them easier to maintain and better machine consumable)
- Used by many Web applications to connect front- and backend systems.

(HTTP + RPC) SOAP
- If you develop heterogeneous distributed systems that need to communicate not only data but also instructions (i.e., method calls).
- Used by many large scale, heterogeneous distributed systems.
A design philosophy for HTTP services:
- **Resources** are the main concept
- **CRUD** (create, read, update, delete) operations on resources should use their corresponding HTTP methods
- Focus on simplicity

**OpenAPI Specification:**
- Creates the RESTful contract for your API.
- RESTful contract describes all resources and their supported methods.
  - a language-agnostic interface description for the RESTful API
- Implemented in, e.g., the **Swagger** framework (see [https://swagger.io/](https://swagger.io/))

No method miss-use like `GET ...publications/?delete_id=42` which is typical for many HTTP services
Service-oriented Middleware
Popular Service Protocols

HTTP
- Used by the largest SOA systems on the planet, e.g., the World Wide Web.

(HTTP) REST
- If you need clearer conventions for HTTP service APIs.
  (e.g. to make them easier to maintain and better machine consumable)
- Used by many Web applications to connect front- and backend systems.

(HTTP + RPC) SOAP
- If you develop heterogeneous distributed systems that need to communicate not only data but also instructions (i.e., method calls).
- Used by many large scale, heterogeneous distributed systems.
An XML-based RPC protocol for making network API requests.

Uses functions as main concepts (in contrast to resources in REST).

Often implemented on top of HTTP but waiving most of its features.

- Comes with its own standards (the web service framework WS[...]).

Idea:

- A server describes the API of its service in a WSDL document (Web Service Description Language; an XML dialect).
- A client can use the WSDL document to generate the API code in its own programming language and then call the API functions.
  - Both server and client can access the API in their own language.

Both programming languages and their IDEs must support SOAP for code and message generation.

- Interoperability without this support is difficult.
Service-oriented Middleware
Simple Object Access Protocol (SOAP)
Simple Object Access Protocol (SOAP)

- Simple example:

```xml
<definitions name="Booking">
  <message name="getBookingRequest">
    <part name="user" type="xs:string"/>
    <part name="house" type="xs:string"/>
  </message>
  <message name="getAvailabilityResponse">
    <part name="available" type="xs:boolean"/>
  </message>
  <portType name="BookingPort">
    <operation name="processBooking">
      <input message="getBookingRequest"/>
      <output message="getAvailabilityResponse"/>
    </operation>
  </portType>
</definitions>
```

A (simple), language-agnostic interface definition

Public interface BookingPort {
  public boolean processBooking(String user, String house);
}
Service-oriented Middleware
Simple Object Access Protocol (SOAP)

- Simple example:

```xml
<?xml version="1.0"?>
<definitions name="Booking">
  <message name="getBookingRequest">
    <part name="user" type="xs:string"/>
    <part name="house" type="xs:string"/>
  </message>

  <message name="getAvailabilityResponse">
    <part name="available" type="xs:boolean"/>
  </message>

  <portType name="BookingPort">
    <operation name="processBooking">
      <input message="getBookingRequest"/>
      <output message="getAvailabilityResponse"/>
    </operation>
  </portType>
</definitions>
```

**WSDL File**

**Binding of an interface to concrete HTTP SOAP calls**

**Service-oriented Middleware**

**Simple Object Access Protocol (SOAP)**

- Simple example:

```xml
<binding name="BookingBinding" type="BookingPort">
  <soap:binding
    style="document"
    transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="processBooking">
    <soap:operation
      soapAction="http://example.com/processBooking"/>
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
</binding>

<service name="BookingService">
  <documentation>A SOAP booking service</documentation>
  <port name="BookingPort" binding="BookingBinding">
    <soap:address location="http://example.com/booking"/>
  </port>
</service>
```

**WSDL File (cont.)**

**Bundling of service calls to a SOAP service**
Overview

Communication

- Message Passing
- OSI Model
- Socket-based Communication
- Message-oriented Middleware
- Service-oriented Middleware
- Database-oriented Middleware
Database-oriented Middleware

Models of Dataflow

Message-Passing Dataflow
- Sending and receiving of messages

Dataflow through Services
- Calling services and waiting for responses

Dataflow through Databases
- Storage and retrieval of data

Process 1 and 2 can be the same (send a message to myself)
Processes write data to and read data from a database:
- Communication through manipulation of (persistent) global state

Requires commonly understood model, schema, and encoding:
- Model: relational, key-value, wide-column, document, graph, ...
- Schema: either schema-on-read or schema-on-write
- Encoding: Unicode, binary, ...

Implicit message exchange:
- No explicit sender or receiver (think of broadcast messages)

Varying message lifetimes:
- Data can quickly be overwritten (= overwritten message is lost).
- Data can stay forever (known as: data outlives code).

Shared memory parallel applications are very similar w.r.t. this model.
Database-oriented Middleware
Models of Dataflow

Databases

Message-Passing

Services

“I have a new booking request!
Someone should handle it …”

“I have a new booking request!
Can you handle it?”

“I have a new booking request!
Book it!”

Distributed Data Management
Communication

Thorsten Papenbrock
Slide 147
Database-oriented Middleware
Models of Dataflow

- Databases
  - Data
  - No response
  - Non-blocking
  - Asynchronous
  - No addressing

- Message-Passing
  - Messages
  - Maybe response
  - Usually non-blocking
  - Usually asynchronous
  - Addressing recipient or queue/mailbox/topic

- Services
  - Function calls
  - Response
  - Blocking
  - Synchronous
  - Addressing recipient

Distributed Data Management
Communication

ThorstenPapenbrock
Slide 148
Communication

Outlook

About what we have seen so far.
