

Verteilte Systeme und Komponenten

Messageorientierte Kommunikation

Martin Bättig

Letzte Aktualisierung: 5. Oktober 2022

FH Zentralschweiz



Inhalt

- Messageorientierte Kommunikation
- Transiente Kommunikation am Beispiel von ZeroMQ und WebSockets.
- Persistente Kommunikation am Beispiel von ActiveMQ Artemis.
- Zusammenfassung

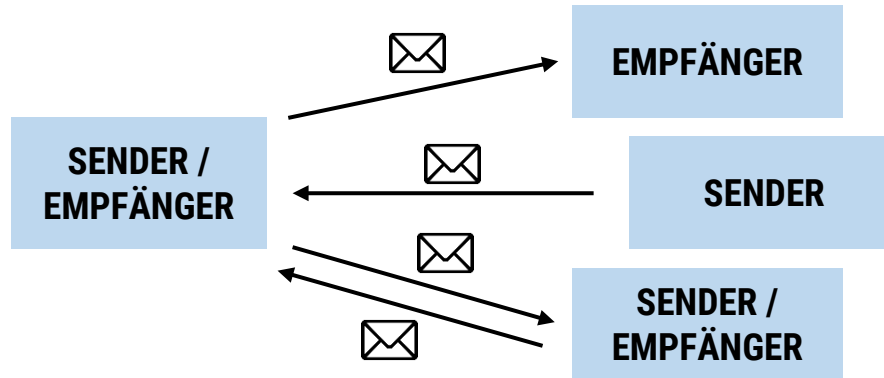
Lernziele

- Sie wissen was Persistenz und Synchronität in der Kommunikation bedeutet und können daraus die möglichen Kommunikationsformen ableiten.
- Sie kennen die verschiedenen Kommunikationsmuster und können einschätzen, wann welches anzuwenden ist.
- Sie wissen welche Rolle Message-Queues und Message-Broker im Rahmen der persistenten Kommunikation einnehmen.

Messageorientierte Kommunikation

Kommunikation mittels Messages (Nachrichten)

Prinzip: Kommunikation mittels **expliziten Messages** gesendet von einem **Sender** zu einem oder mehreren **Empfängern**.



Verwendung:

- Nebenläufige und parallele Programmierung (z.B. Actor-Model [1]).
- Interprozesskommunikation (z.B. Unix-Sockets).
- Kommunikation zwischen verteilten Systemen.

Abgrenzung zu:

- Synchroner Remote-Procedure-Call (Transparenz)
- Streaming (z.B. reines TCP, Unix-Pipes).

[1] Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A universal modular ACTOR formalism for artificial intelligence. In Proceedings of the 3rd international joint conference on Artificial intelligence (IJCAI'73).

Fragestellungen der messageorientierten Kommunikation

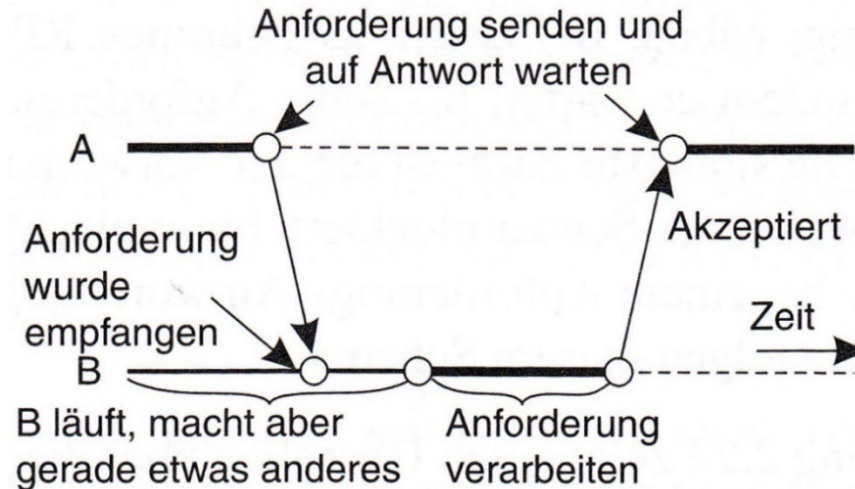
- **Zuverlässigkeit:** Können Messages verloren gehen?
- **Reihenfolge:** Kommen die Messages in der vorgesehenen Reihenfolge an?
- **Anzahl Empfänger:**
 - Unicast / Anycast (1:1)
 - Multicast / Broadcast (1:n)
 - Client-Server (n:1)
 - Peer-to-Peer (m:n)
- **Aufführungszeitpunkt:** Synchron vs. asynchron.
- **Sicherheit:** Offen vs. Zugangsgeschützt, Verschlüsselung.
=> Ggf. Bereits auf Stufe Kommunikationskanal klären.

Persistente vs. transiente Kommunikation

- **Persistente** Kommunikation: Nachricht wird solange gespeichert bis Empfänger bereit ist (z.B. E-Mail).
- **Transiente** Kommunikation: Nachricht wird nur gespeichert, solange sendende und empfangende Applikation ausgeführt werden (z.B. Router, Socket).

Transiente messageorientierte Kommunikation

Synchrone Kommunikation

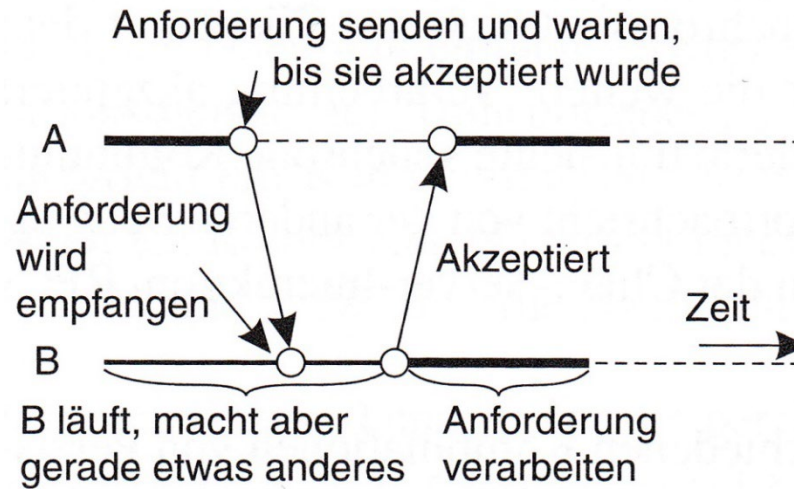


Beispiele:

- Remote-Procedure-Call.
- Anfrage einer Ressource mittels HTTP-Protokoll und synchroner Auslieferung.

Zeit bis zur Verarbeitung der Anforderung kurz halten, z.B. mit Threads oder Non-Blocking I/O.

Asynchrone Kommunikation



Beispiele:

- Asynchroner Remote-Procedure-Call.
 - Aber wie kommt das Resultat zurück?
- Anfrage einer Ressource mittels HTTP-Protokoll mit Quittierung (ACK)
 - Auch hier: Wie kommt der Client an das Resultat der Anfrage?

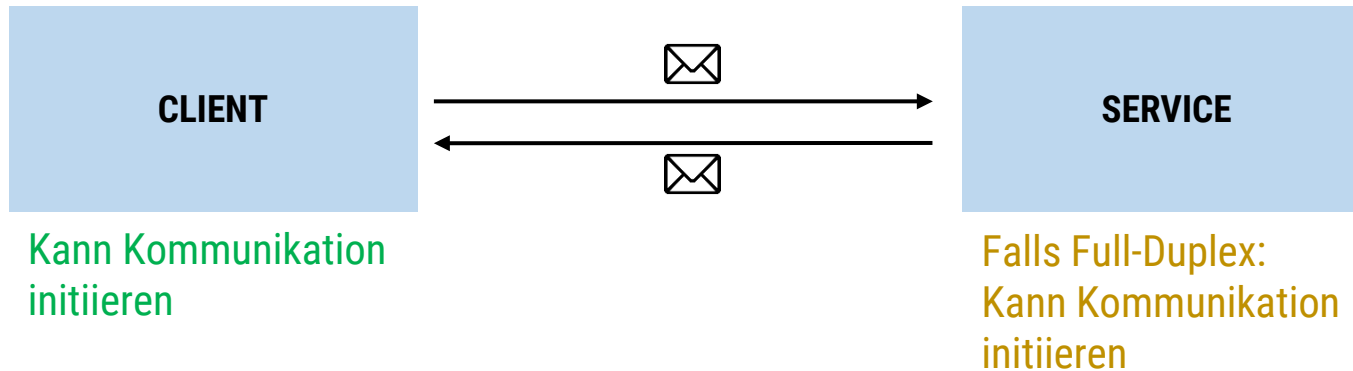
Kommunikationsmuster (Auswahl)

- Kommunikation lässt sich in Muster einteilen.
- Bekannte Muster sind:
 - **Request-Reply:** Client-Server (One-To-One).
 - **Publish-Subscribe:** Datenverteilung (One-To-Many).
 - **Pipeline:** Verarbeitung in mehreren Schritten (One-To-Many).

Muster: Request-Reply

Ziel: Verbinden **einer Menge von Clients** mit **einer Menge von Services**.

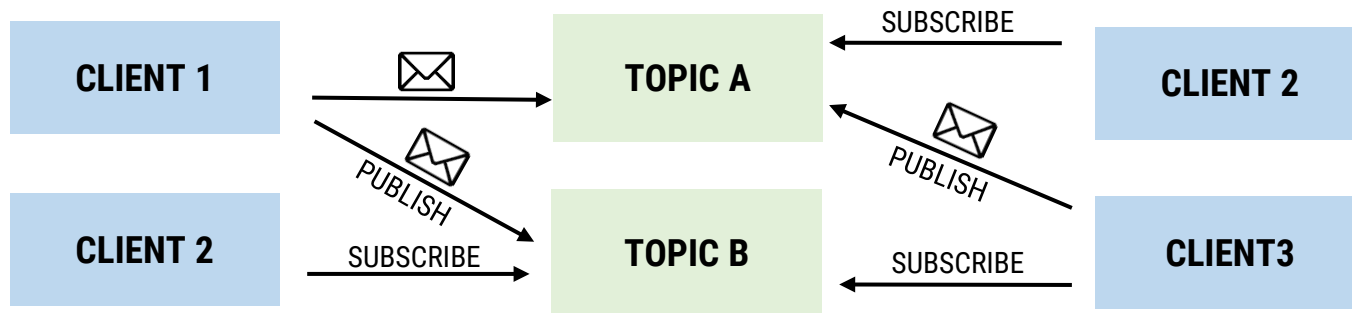
- Half-Duplex: Nur Client kann Nachrichten initiieren.
- Full-Duplex: Sowohl Client als auch Service können Nachrichten initiieren.



Publish-Subscribe Muster

Ziel: Datenverteilung.

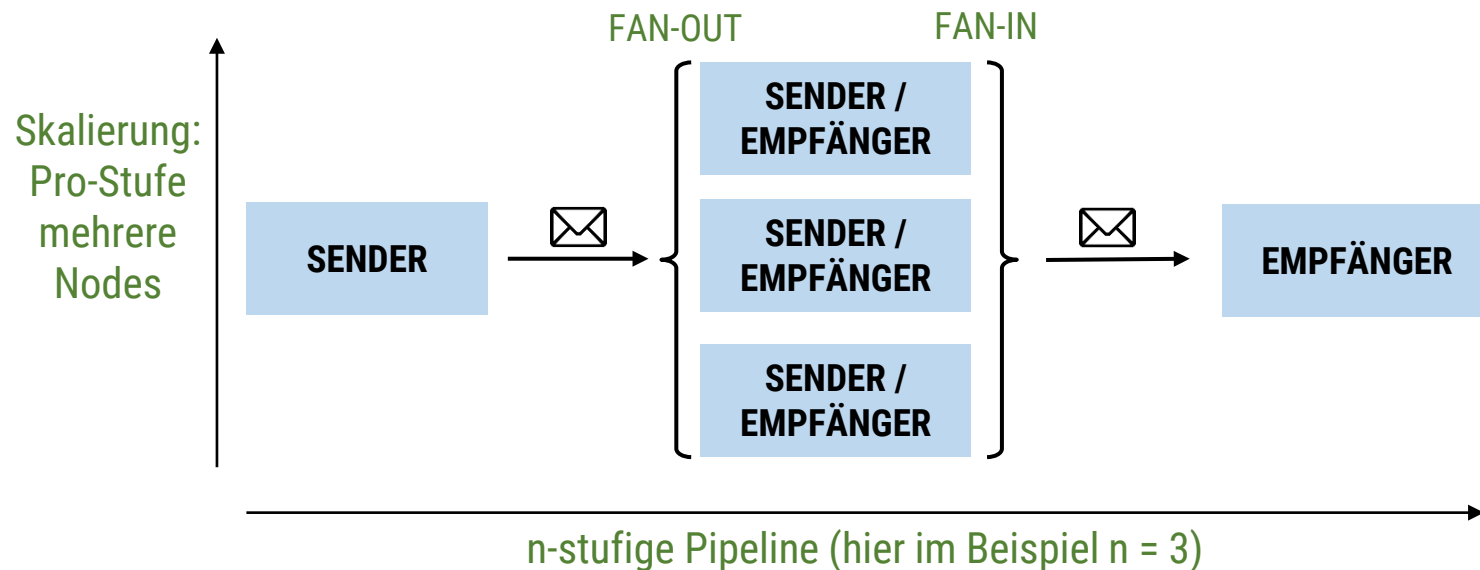
- Verbindet eine **Menge von Publishers** mit einer **Menge von Subscribers**.
- Verbreitet in Enterprise- (ActiveMQ/Websphere) und IoT-Umfeld (-> MQTT).



Pipeline-Muster

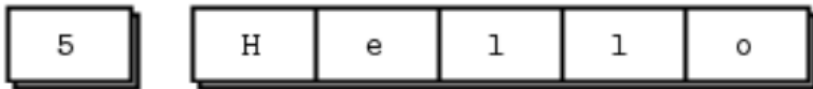
Ziel: Aufgabenverteilung

- Verbindet mehrere Nodes in einem Fan-out / Fan-in Pattern.
- Das Pattern kann **mehrere Stufen** und **sogar Schleifen** haben.
- Auch bekannt als parallele Aufgabenverteilung und -zusammenführung.



Technologie: ZeroMQ

- Library für messageorientierte Kommunikation.
- Messages werden i.d.R. asynchron gesendet.
- Bekanntes Socket-Prinzip auf höherem Abstraktionsniveau.
 - Spezifische Sockets für bestimmte Kommunikationsmuster.
- Messageinhalt beliebig (bytes oder Text).
- Framing basierend auf Länge



ZeroMQ: Sockets passend zu Kommunikationsmustern.

Pattern	Socket	Einsatzgebiet
Request-Reply	REQ	Sendet Anfrage (an RES-Socket). Half-Duplex.
	RES	Beantwortet Anfragen (von REQ-Sockets). Half-Duplex.
PubSub	PUB	Publiziert Message unter bestimmtem Topic.
	SUB	Empfängt Messages für abonnierte Topics.
Pipeline	PUSH	Sendet Message an Verarbeiter (PULL-Socket).
	PULL	Empfängt Message zur Verarbeitung (von PUSH-Socket).

Auswahl: ZeroMQ kennt weitere Sockets.

ZeroMQ Beispiel 1: Half-Duplex Echo (Client)

```
public static final String ADDRESS = "tcp://localhost:5555";

public static void main(String[] args) {
    try (ZContext context = new ZContext()) {
        LOG.info("Echo client connecting to " + ADDRESS);
        ZMQ.Socket socket = context.createSocket(SocketType.REQ);

        socket.connect(ADDRESS);

        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        while(true) {
            String input = in.readLine();
            socket.send(input.getBytes(ZMQ.CHARSET));
            byte[] reply = socket.recv();
            LOG.info("Received " + new String(reply, ZMQ.CHARSET));
        }
    } catch (IOException ex) {
        LOG.error(ex.getMessage());
    }
}
```

REQ: Socket für Requests

Clients benutzen connect

REQ/REP: Auf jedes send muss ein recv folgen

ZeroMQ Beispiel 1: Half-Duplex Echo (Server)

```
public static final String ADDRESS = "tcp://localhost:5555";
```

```
public static void main(String[] args) {  
    try (ZContext context = new ZContext()) {
```

```
        ZMQ.Socket socket = context.createSocket(SocketType.REP);
```

REP: Socket für Replies

```
        socket.bind(ADDRESS);
```

Server benutzen bind

```
        while (true) {
```

```
            byte[] reply = socket.recv();  
            LOG.info("Received message " + new String(reply, ZMQ.CHARSET));  
            socket.send(reply);  
        }
```

REQ/REP: Auf jedes recv muss ein send folgen

```
    }  
}
```

ZeroMQ Beispiel 2: Data Distribution (Publisher)

```
public static final String ADDRESS = "tcp://localhost:5555";
```

```
public static void main(String[] args) {  
    try (ZContext context = new ZContext()) {  
        ZMQ.Socket socket = context.createSocket(SocketType.PUB);
```

PUB: Socket für Publisher

```
        socket.bind(ADDRESS);
```

Publisher benutzen bind

```
        while (true) {  
            long station = (long) Math.floor(Math.random() * 3.0);  
            long temp = 15 + (long) Math.floor(Math.random() * 10.0);
```

```
            String message = "Temp/" + station + "/" + temp;
```

```
            socket.send(message.getBytes(ZMQ.CHARSET));  
            LOG.info("Published '" + message + "'");
```

Topic implizit definiert
durch Message-Prefix

```
            Thread.sleep(1000);
```

```
        }  
    } catch (InterruptedException e) {  
        LOG.info("interrupted"); // cannot occur  
    }  
}
```

ZeroMQ Beispiel 2: Data Distribution (Subscriber)

```
public static final String ADDRESS = "tcp://localhost:5555";
```

```
public static void main(String[] args) {  
    String topic = args[0];
```

SUB: Socket für Subscriber

```
    try (ZContext context = new ZContext()) {  
        ZMQ.Socket socket = context.createSocket(SocketType.SUB);
```

```
        socket.connect(ADDRESS);
```

Subscriber benutzen bind

```
        socket.subscribe(topic);
```

Bestimmten Topic abonnieren
(Message-Prefix)

```
        while (true) {
```

```
            byte[] message = socket.recv();
```

Nur Messages der
abonnierten Topics

```
            LOG.info("Received: " + new String(message, ZMQ.CHARSET));
```

```
        }
```

```
    }
```

```
}
```

ZeroMQ Beispiel 3: Taskverarbeitung (Producer)

```
public static final String ADDRESS = "tcp://localhost:5555";
```

```
public static void main(String[] args) {  
    try (ZContext context = new ZContext()) {  
        LOG.info("Providing tasks on " + ADDRESS);
```

```
        // Socket to talk to server
```

```
        ZMQ.Socket socket = context.createSocket(SocketType.PUSH);
```

PUSH: Socket für Producer

```
        socket.bind(ADDRESS);
```

bind für Fan-out
connect für Fan-in

```
        int i = 0;
```

```
        while(true) {
```

```
            String workPackage = "Work Package #" + ++i;
```

```
            socket.send(workPackage.getBytes(ZMQ.CHARSET));
```

```
            LOG.info("Send task '" + workPackage + "'");
```

```
            Thread.sleep(1000);
```

```
        }
```

```
    } catch (InterruptedException e) {
```

```
        LOG.info("interrupted"); // cannot occur
```

```
    }
```

```
}
```

Blockiert bis Consumer
verbunden sind

ZeroMQ Beispiel 3: Taskverarbeitung (Consumer)

```
public static final String ADDRESS = "tcp://localhost:5555";

public static void main(String[] args) {
    try (ZContext context = new ZContext()) {
        LOG.info("Listening for tasks on " + ADDRESS);

        // Socket to talk to server
        ZMQ.Socket socket = context.createSocket(SocketType.PULL);
        socket.connect(ADDRESS);

        while(true) {
            byte[] bytes = socket.recv();
            LOG.info("Received task '" + new String(bytes, ZMQ.CHARSET) + "'");
            Thread.sleep(3000);
            LOG.info("Processed task '" + new String(bytes, ZMQ.CHARSET) + "'");
        }

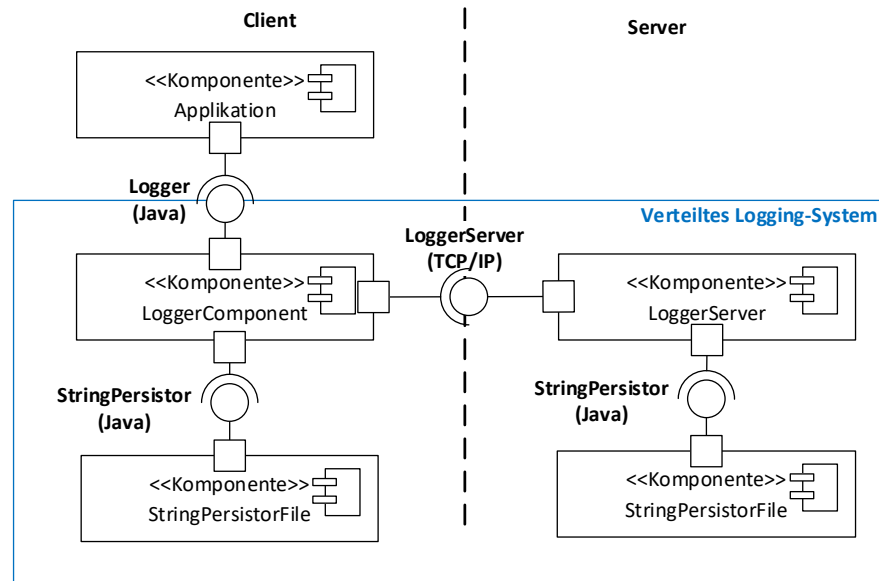
    } catch (InterruptedException e) {
        LOG.error("interrupted"); // cannot occur
    }
}
```

PULL: Socket für Consumer

Connect für Fan-out
Bind für Fan-in

Diskussion: Kommunikationsmuster

Diskutieren Sie zu zweit, ob und wo Sie Kommunikationsmuster im Logger verwenden (können). Sehen Sie Vorteile/Nachteile?



Angenommen ein oder mehrere Logger-Viewer, welche Log-Messages in Real-Time anzeigen sollen, sollen an das verteilte Logger-System via TCP/IP angebunden werden. Wie sieht die Situation dann aus?

Wir besprechen die Resultate im Plenum.

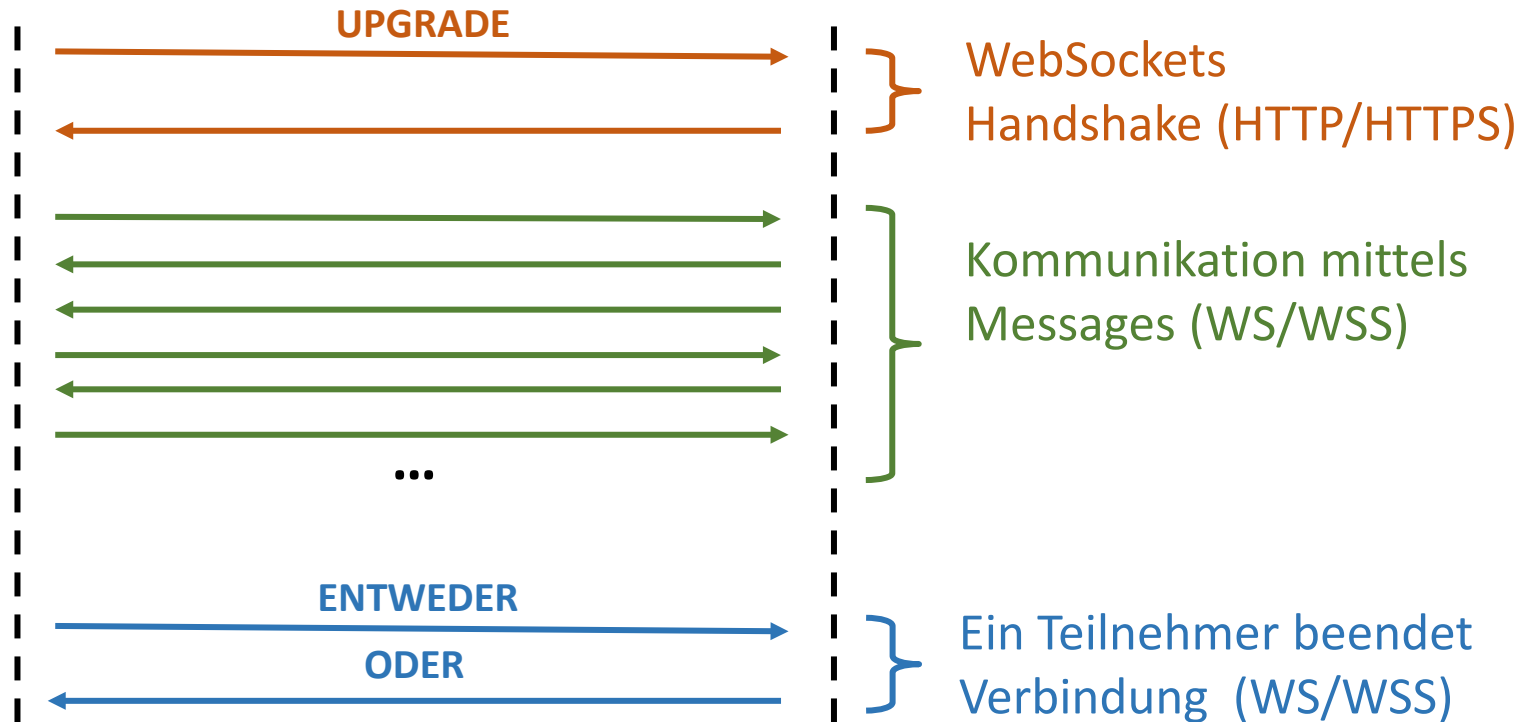
Protokoll: WebSockets

- Full-Duplex-Kommunikation.
- Protokoll auf Anwendungslevel (KEIN Transportprotokoll wie TCP/UDP).
- Basiert auf HTTP -> Protokollupgrade.
- RFC 6455 <https://tools.ietf.org/html/rfc6455>
- Viele Implementationen: Tomcat, Jetty, Nginx, Apache, usw.

WebSockets: Kommunikationsverlauf

Client

Server



WebSockets: Handshake

Client

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: x3JJHbDL1EzLkh9GBhXDw==
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
Origin: http://example.com
```

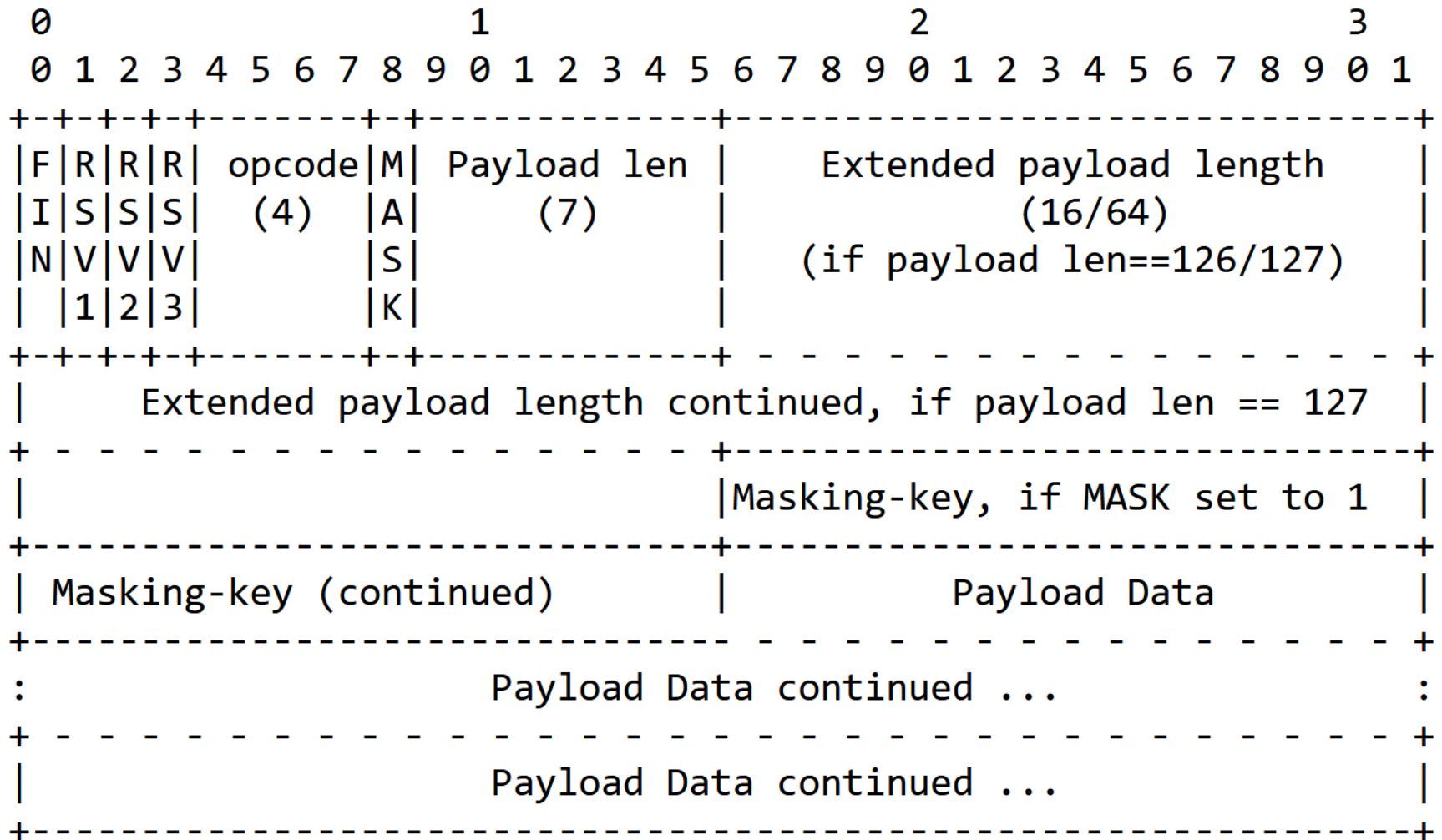
Server

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: HSmrc0sMlYUkAGmm5OPpG2HaGWk=
Sec-WebSocket-Protocol: chat
```

Quelle: <https://www.rfc-editor.org/rfc/rfc6455>

WebSocket Transport

Binärprotokoll: Messages übermittelt via TCP mittels Frames.



Quelle: <https://www.rfc-editor.org/rfc/rfc6455>

WebSockets Server Endpoint API (Beispiel Jetty Server API)

`org.eclipse.jetty.websocket.api.WebSocketListener`

Methode	Beschreibung
void onWebSocketConnect(Session session)	Client hat sich verbunden.
void onWebSocketClose(int statusCode, String reason)	Session wurde clientseitig geschlossen.
void onWebSocketBinary(byte[] payload, int offset, int length)	Binärdaten empfangen.
void onWebSocketText(String message)	Textnachricht empfangen (vollständige Message in String enthalten).
void onWebSocketError(Throwable cause)	Fehler aufgetreten.

Beispiel: Chat-Server mit WebSockets (Session Manager)

```
public class ChatSessions {  
    private final List<Session> sessions = new LinkedList<>();  
  
    public synchronized void add(Session session) {  
        sessions.add(session);  
    }  
    public synchronized void remove(Session session) {  
        sessions.remove(session);  
    }  
  
    public synchronized void broadcast(String message) {  
        Iterator<Session> it = sessions.iterator();  
        while (it.hasNext()) {  
            Session session = it.next();  
            try {  
                session.getRemote().sendString(message);  
            } catch (IOException e) {  
                it.remove();  
                LOG.error("removed session from broadcast list«  
                    + session.getRemoteAddress() + " due to IOException");  
            }  
        }  
    }  
}
```

Alle Operationen auf Sessionliste sind Thread-safe


Sende Nachricht an Session

Beispiel: Chat-Server mit WebSockets (Endpoint)

```
public class ChatEndPoint implements WebSocketListener {
    private static final ChatSessions CHAT_SESSIONS = new ChatSessions();
    private Session session;
    private String username;

    public void onWebSocketConnect(Session session) {
        this.session = session;
        CHAT_SESSIONS.add(session);
    }
    public void onWebSocketClose(int statusCode, String reason) {
        CHAT_SESSIONS.remove(session);
    }
    public void onWebSocketError(Throwable cause) {
        CHAT_SESSIONS.remove(session);
        LOG.error("web socket error occurred" , cause);
    }
    public void onWebSocketText(String message) {
        if (username == null) {
            username = message;
        } else {
            CHAT_SESSIONS.broadcast(username + ": " + message);
        }
    }
}
```

Zustandsbasiertes-
Protokoll



Beispiel: Chat-Server mit WebSockets (Einbettung in Jetty)

Beispiel zur Illustration

```
private static final int PORT = 8080;

public static void main(String[] args) throws Exception {
    Server server = new Server(PORT);
    ServletContextHandler servletContextHandler = new ServletContextHandler();
    server.setHandler(servletContextHandler);

    Servlet websocketServlet = new JettyWebSocketServlet() {
        @Override protected void configure(JettyWebSocketServletFactory factory) {
            factory.setIdleTimeout(Duration.ofMinutes(30));
            factory.addMapping("/", (req, res) -> new ChatEndPoint());
        }
    };
    servletContextHandler.addServlet(new ServletHolder(websocketServlet), "/chat");
    JettyWebSocketServletContainerInitializer.configure(servletContextHandler, null);

    server.start();
    LOG.info("Chat server listening on port " + PORT);
}
```

Übung: Chatserver mit WebSockets

Der Chatserver interpretiert die erste Message als Benutzernamen und alle nachfolgenden Messages als Chatnachrichten.

Fragen:

- Ist dieses Protokoll robust? Warum?
- Wie müsste man das Protokoll ändern um explizite IDs hinzufügen?
- Vorteile/Nachteile?

Der Chatserver sendet beim Eintreffen einer Nachricht eines Benutzers eine Push-Nachricht an alle Benutzer.

Fragen:

- In welchen Szenarios ist dieses Push-Verfahren besser als ein Pull-Verfahren (Client fragt nach, ob neue Nachrichten da sind)?
- Könnte es mit dem Push-Verfahren Probleme geben, welche?

Besprechen Sie diese Fragen zu zweit (oder dritt) während 10 Minuten. Wir diskutieren anschliessend im Plenum.

Persistente Messageorientierte Kommunikation

Zuverlässige Messagetransfer

Zuverlässige Auslieferung: Eine Message wird ausgeliefert und zwar exakt einmal.

Unzuverlässige Auslieferung: Eine Message wird entweder gar nicht oder mehr als einmal ausgeliefert.

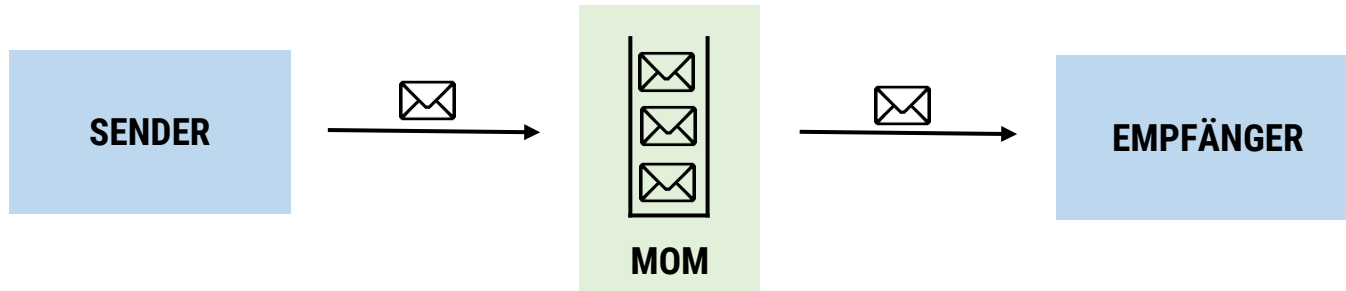
Ursachen:

- Messages über unzuverlässigen Kanal gesendet.
- Messages verworfen (Queue voll).
- Systeme laufen nicht immer.
- Prozesse oder Systeme können während Verarbeitung ausfallen.

Persistente messageorientierte Kommunikation

Message-Oriented-Middleware (MOM):

- Zwischenspeicherung von Messages.
- Zeitversetzte Kommunikation.



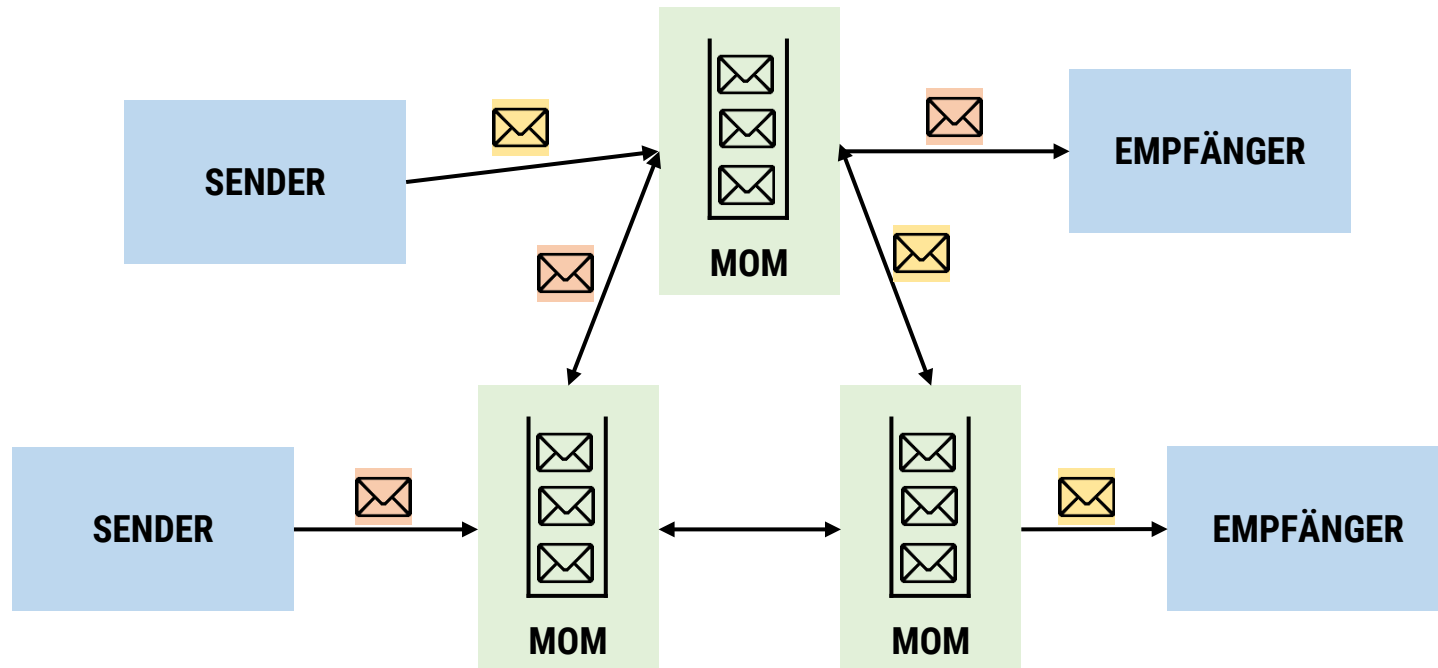
- Sender oder Empfänger müssen nicht aktiv sein während Übertragung.
- Unterstützt Transfers, welche Minuten dauern (anstelle von ms).

Populäre MOMs:

- Apache ActiveMQ/ ActiveMQ Artemis
- RabbitMQ
- Websphere MQ

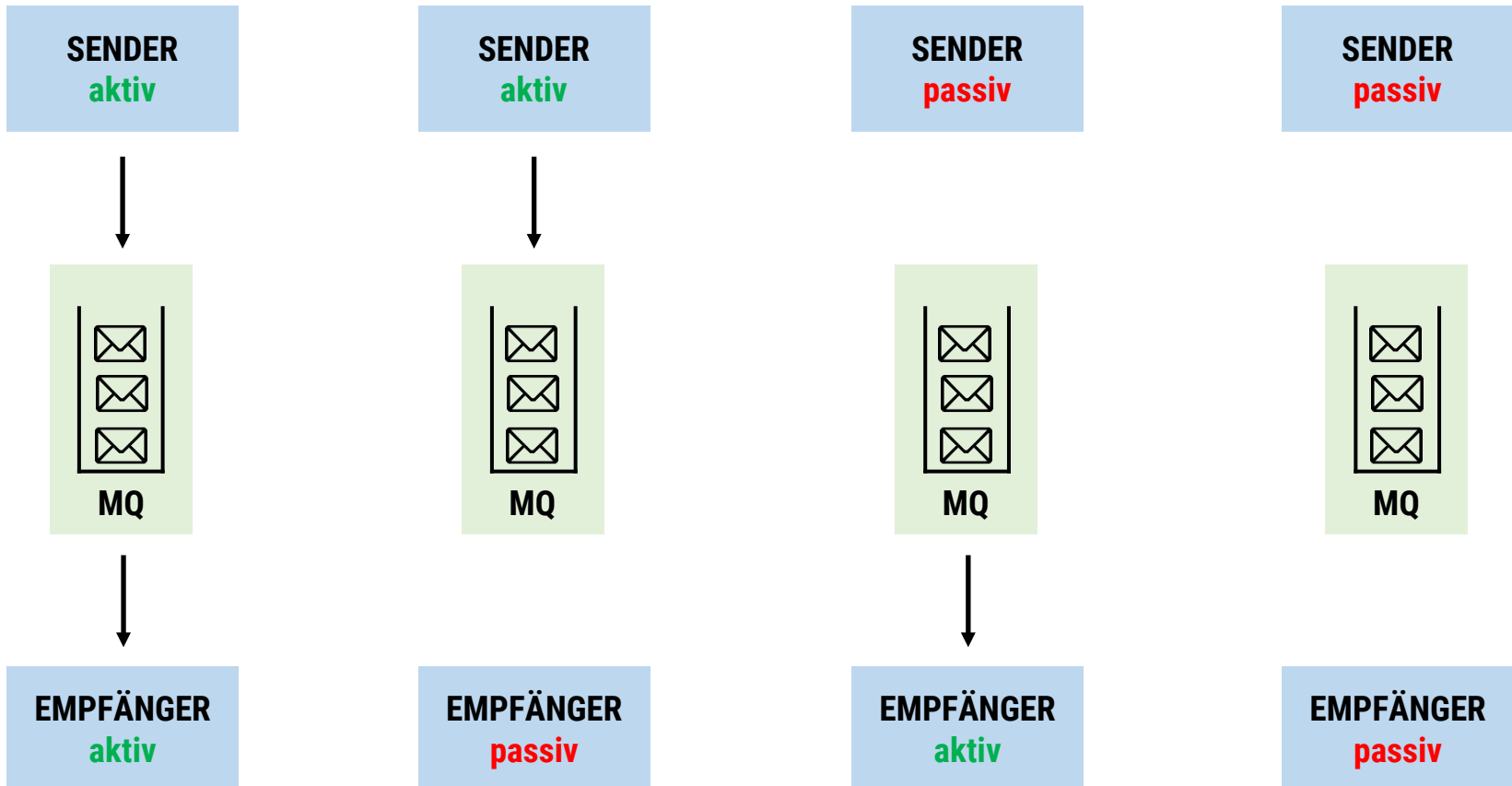
Message-Queueing-Model

- Kommunikation mittels Einfügen von Messages in spezifische Warteschlange.
- Keine Garantien, **wann** eine Message gelesen wird.
- Message werden von Kommunikationsservern weitergeleitet, bis zum Ziel.
 - Oft sind die Kommunikationsserver direkt miteinander verbunden.
- Typischerweise eine Queue pro Kommunikationsteilnehmer.
 - Queue kann aber geteilt werden.



Zeitliche Entkopplung mittels Message-Queues

Vier Kombinationen:



Typisches Message-Queue-API

Operation	Beschreibung
<code>put()</code>	Füge eine Message einer bestimmten Queue hinzu.
<code>get()</code>	Warte bis eine Queue nicht mehr leer ist und retourniere die erste Message in der Queue. => blockierend
<code>poll()</code>	Überprüfe ob eine Queue mindestens eine Message enthält. Ist dies der Fall ist, retourniere die erste Message in der Queue. => nicht blockierend
<code>notify(callback)</code>	Registriere eine Callback-Funktion, welche aufgerufen wird, sobald eine Message in dieser Queue eintrifft.

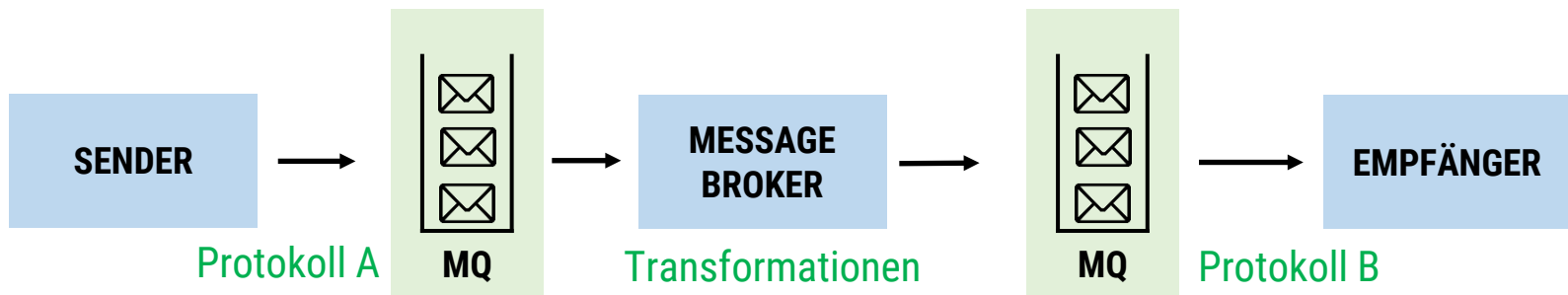
Populäre Message-Queuing-Protokolle

- AMQP: Ubiquitous, secure, reliable and open internet protocol for handling business messaging.
- MQTT: light weight, client to server, publish / subscribe messaging protocol.
Oft im IoT-Bereich verwendet.
- STOMP: text-orientated wire protocol.
- XMPP: eXtensible Messaging and Presence Protocol.
- [OpenWire: Proprietäres Protokoll von ActiveMQ.]

Message-Broker

Ziel: Integration von heterogenen Applikationen.

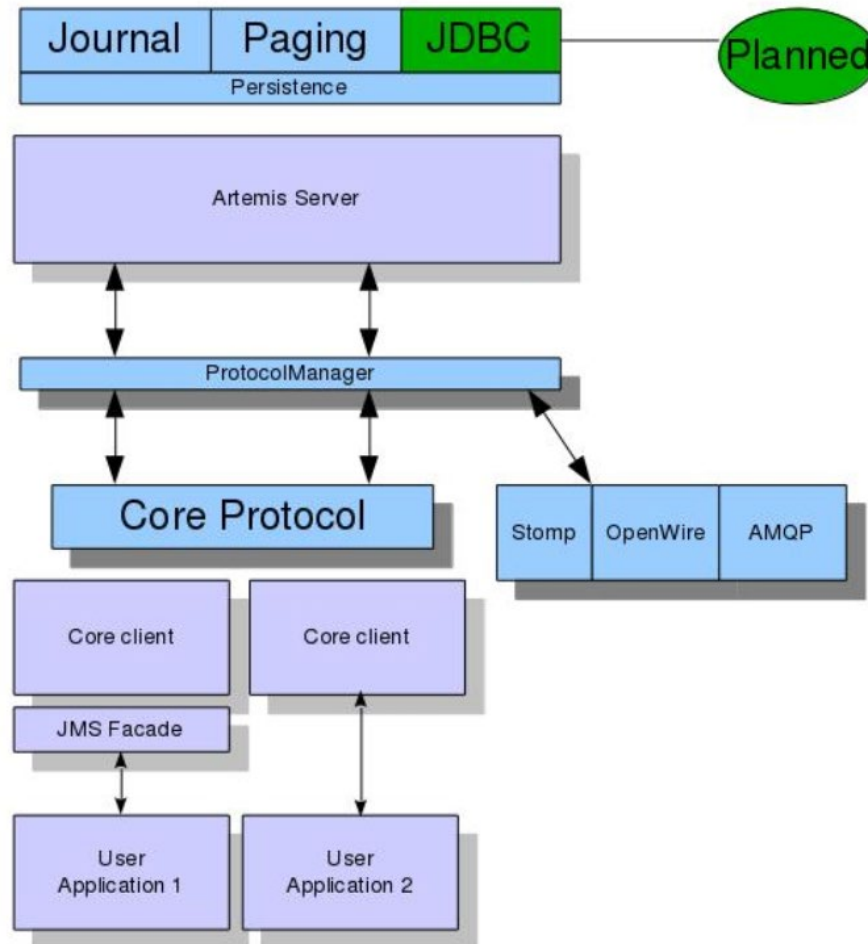
- System mit verschiedene Protokollen und Nachrichtenformaten verbinden.
- Alles auf den gemeinsamen Nenner bringen hätte Komplexität $N \times N$.
- Broker wandeln zwischen Protokollen und Nachrichtenformaten und routen Messages zu verschiedenen Ziel (ggf. mit Publish-Subscribe).



Konzeptionell auf gleicher Stufe wie Sender oder Empfänger, oft aber direkt mit einem MQ-System verknüpft.

Fallbeispiel: Apache ActiveMQ Artemis

- open source, multi-protocol, Java-based message broker
- <https://activemq.apache.org/>
- **Architektur:**



Quelle: <https://activemq.apache.org/>

Fallbeispiel: Broker und Queue erstellen, Admin-Interface

Neuen Broker erstellen

```
artemis create mybroker --user admin --password admin --require-login
```

Broker starten

```
mybroker\bin\artemis run
```

Neue Queue erstellen

```
mybroker\bin\artemis queue create --user admin --password admin --address  
demoQueue --anycast --name demoQueue --auto-create-address --durable --  
preserve-on-no-consumers
```

Webinterface

```
http://localhost:8161
```

Fallbeispiel: ActiveMQ Artemis Producer

```
public static void main(String args[]) throws Exception {
    ServerLocator locator =
        ActiveMQClient.createServerLocator(ARTEMIS_LOCATION);
    ClientSessionFactory factory = locator.createSessionFactory();
    ClientSession session = factory.createSession(
        "admin", "admin", true, false, false, false, 1);

    ClientProducer producer = session.createProducer("demoQueue");

    BufferedReader userIn = new BufferedReader(new InputStreamReader(System.in));
    while (true) {
        session.start();
        ClientMessage message = session.createMessage(true);
        message.getBodyBuffer().writeString(userIn.readLine());
        producer.send(message);
        LOG.info("send message: " + message.getBodyBuffer().readString());
        session.commit();
    }
}
```

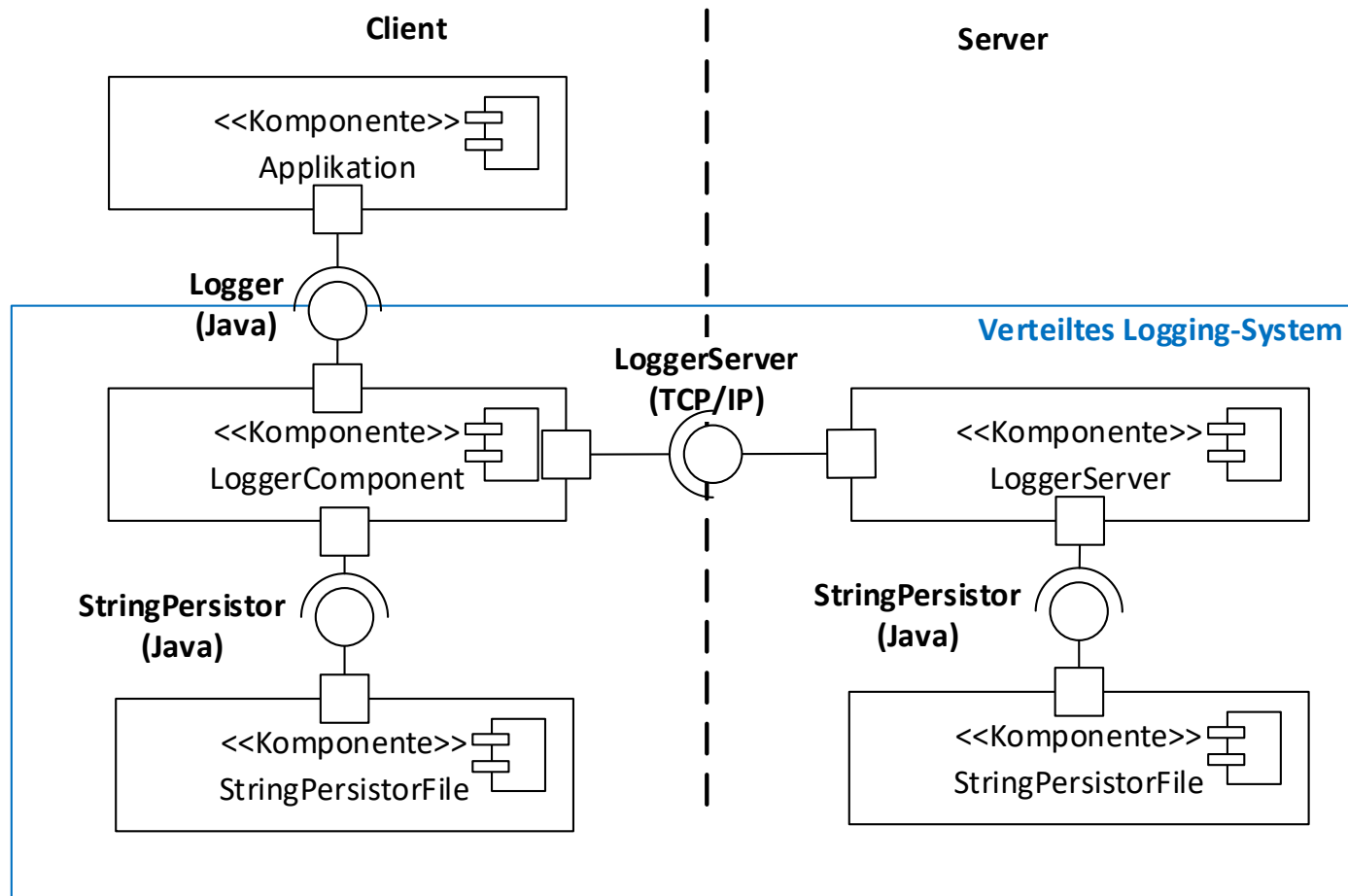
Fallbeispiel: ActiveMQ Artemis Consumer

```
public static void main(String args[]) throws Exception {
    ServerLocator locator =
        ActiveMQClient.createServerLocator(ARTEMIS_LOCATION);
    ClientSessionFactory factory = locator.createSessionFactory();
    ClientSession session = factory.createSession(
        "admin", "admin", true, false, false, false, 1);

    ClientConsumer consumer = session.createConsumer("demoQueue");

    while(true) {
        session.start();
        ClientMessage message = consumer.receive();
        LOG.info("received msg: " + message.getBodyBuffer().readString());
        session.commit();
    }
}
```

Diskussion: Wann persistente Kommunikation einsetzen?



Zusammenfassung

- Für das Design einer messageorientierten Kommunikation sind verschiedene Anforderungen festzulegen: Form der Nachrichten, Art der Kommunikation (transient, persistent, asynchron, synchron).
- Verschiedene Kommunikationsmuster (Request-Reply, Publish-Subscribe, Pipeline) bestimmen die Architektur einer Applikation.
- Middleware (Beispiel: ZeroMQ, WebSockets) unterstützen beim Implementieren dieser Kommunikationsmuster.
- Persistente Message-Queues ermöglichen zeitliche Entkopplung von Komponenten oder Systemen.
- Message-Broker ermöglichen Integration heterogener Systeme.

Literatur

- Distributed Systems (3rd Edition), Maarten van Steen, Andrew S. Tanenbaum, Verleger: Maarten van Steen (ehemals Pearson Education Inc.), 2017.
- ZeroMQ, by Pieter Hintjens, O'Reilly Media, Inc., 2013.
- Apache ActiveMQ Artemis 2.26.0 User Manual,
<https://activemq.apache.org/components/artemis/documentation/>

Fragen?