Tema 6: Integración de Aplicaciones

Message-Oriented middleware (MOM)

WebSockets

Sistemas de Información para Internet
3º del Grado de Ingeniería Informática (tres menciones)

Departamento de Lenguajes y Ciencias de la Computación
Escuela Técnica Superior de Ingeniería Informática
Universidad de Málaga

Message-oriented middleware (MOM)

Message-Oriented middleware (MOM)

- Es una infraestructura de software que admite el envío y recepción de mensajes asíncronos entre sistemas distribuidos.
- Buffer entre sistemas que producen y consumen información cada uno a un ritmo.
- Los productores y consumidores no se conectan directamente, ni siquiera saben una de otra.

Message-Oriented middleware (MOM)

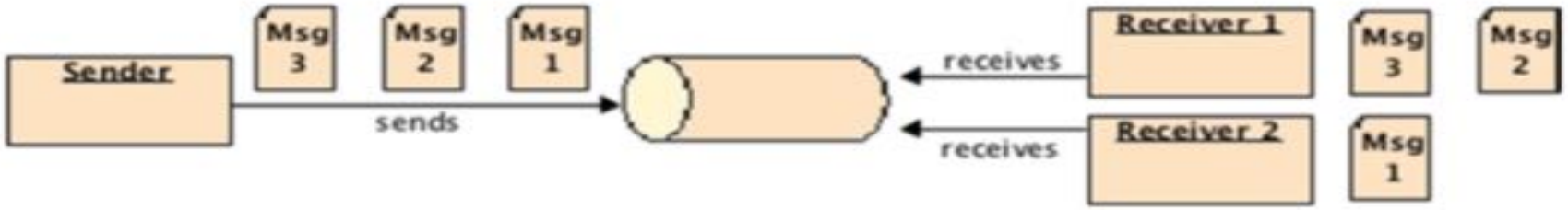
- Se encarga de que todos los mensajes lleguen siempre a su destino.
- La comunicación entre emisor y receptor es asíncrona, y en ningún momento están directamente conectados

Message-Oriented middleware (MOM)

- El emisor envía el mensaje y no se queda a la espera de recibir confirmación de recepción de su mensaje, sino que sigue trabajando normalmente.
- Hay dos modelos de este tipo de middleware:
  - Punto a punto
  - Publicación/suscripción

Message-Oriented middleware (MOM)
Punto a punto

- Los mensajes van dirigidos a un único receptor.
- El mensaje queda almacenado en una **cola** hasta el que el receptor quiera o pueda recogerlo.
- Si hubiesen varios receptores no obtendrían todos los mensajes.

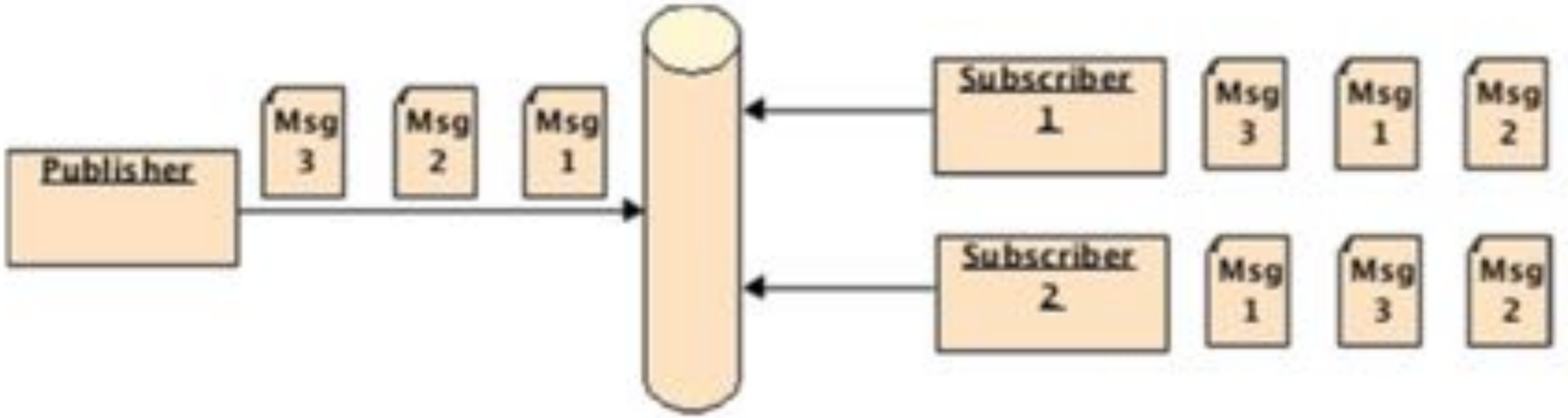


Message-Oriented middleware (MOM) Publicación / suscripción

- Están formados por dos actores:
  - Emisores o productores de información.
  - Consumidores de dicha información.
- Los Emisores: Envían mensajes al MOM, que se encarga de entregarlos a los suscriptores correspondientes.
- Los Consumidores: Pueden suscribirse a un determinado tipo de mensajes que tengan relación con un tema concreto o responda a un patrón determinado.

Message-Oriented middleware (MOM) Publicación / suscripción

- Varios consumidores pueden obtener el mensaje.



Message-Oriented middleware (MOM) Publicación / suscripción

- El suscriptor debe permanecer continuamente activo para recibir mensajes, a menos que establezca una suscripción duradera.
- Duradera: los mensajes publicados mientras el suscriptor no está conectado se redistribuyen cada vez que se vuelve a conectar.

Message-Oriented middleware (MOM) Estructura mensaje

- Un mensaje está formado por:
  - Cabecera: contiene información para identificar y enrutar el mensaje
  - Propiedades: pares nombre-valor. Se utiliza para filtrar
  - Cuerpo: contiene el contenido en diferentes formatos

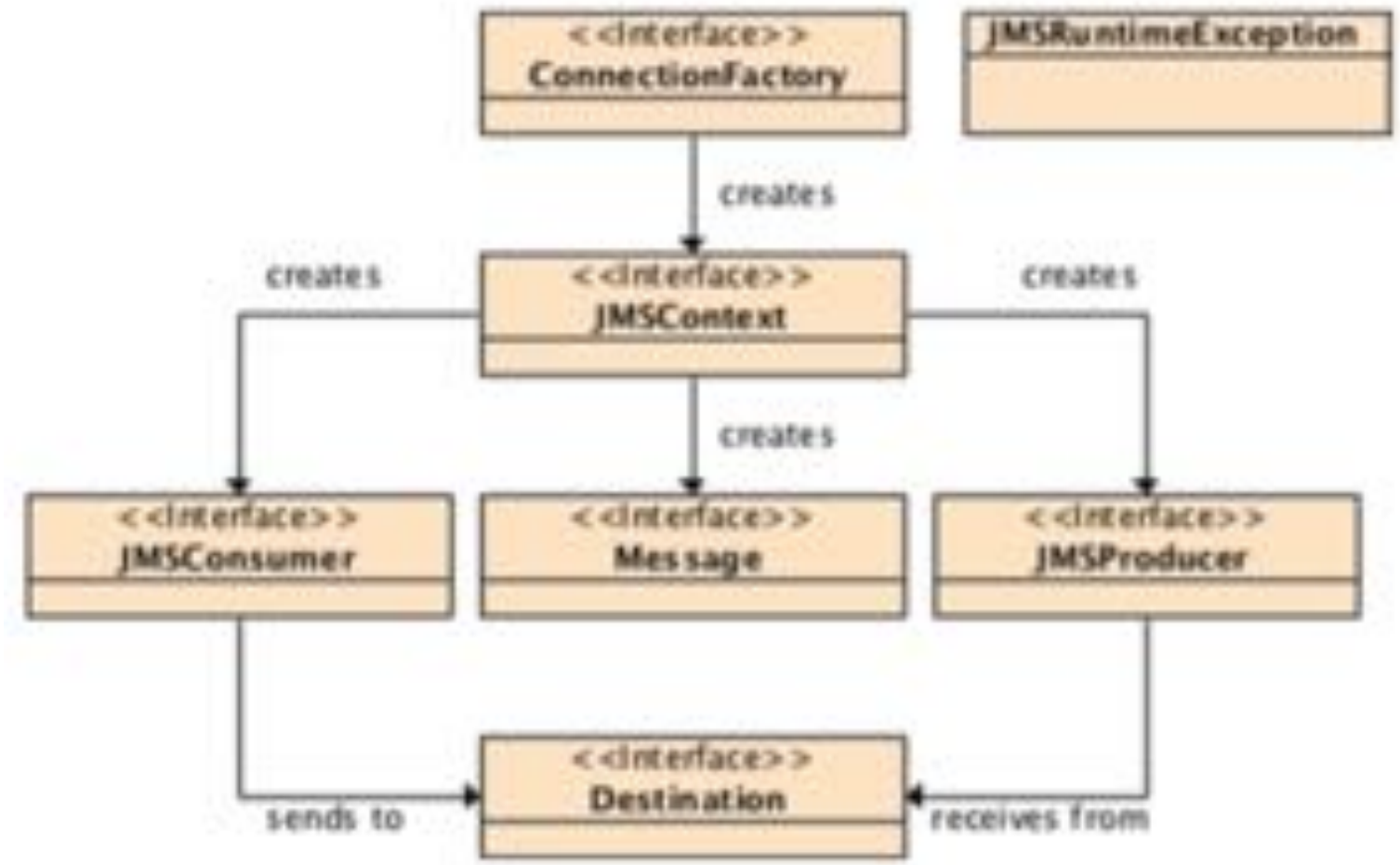
Message-Oriented middleware (MOM) JMS

- **Proveedor JMS**: JMS es un API, necesita un proveedor que la implemente.
- **Cliente JMS**: Una aplicación o proceso que genera y/o recibe mensajes.
- **Productor JMS**: Cliente JMS que envía mensajes.
- **Consumidor**: Cliente JMS que recibe mensajes.

Message-Oriented middleware (MOM) JMS

- **Mensajes JMS**: Objeto que contiene los datos que se transfieren entre clientes JMS
- **Cola JMS (*queue*)**: Contiene los mensajes que se envían y están a la espera de ser leídos, FIFO
- **Tema JMS (*topic*)**: Mecanismo de distribución para publicar mensajes que se entregan a varios suscriptores.

Message-Oriented middleware (MOM)
Java Messaging Service API



Message-Oriented middleware (MOM) Java Messaging Service API

- Tres interfaces principales:
  - **JMSContext**: conexión activa a un JMS provider y un contexto monohilo para enviar y recibir mensajes
  - **JMSProducer**: objeto creado por JMSContext para enviar mensajes desde una cola o tópico
  - **JMSConsumer** : objeto creado por JMSContext para recibir mensajes desde una cola o tópico

Message-Oriented middleware (MOM) Java Messaging Service API

- Creación de un productor en un EJB

```
@Stateless
public class ProducerEJB {

  @Resource(lookup = "jms/javaee7/ConnectionFactory")
  private ConnectionFactory connectionFactory;
  @Resource(lookup = "jms/javaee7/Queue")
  private Queue queue;

  public void sendMessage() {

    try (JMSContext context = connectionFactory.createContext()) {
      context.createProducer().send(queue, "Text message sent at " + new Date());
    }
  }
}
```

Message-Oriented middleware (MOM)
Java Messaging Service API

```
public class Listener implements MessageListener {

  public static void main(String[] args) {

    try {
      // Gets the JNDI context
      Context jndiContext = new InitialContext();

      // Looks up the administered objects
      ConnectionFactory connectionFactory = (ConnectionFactory) ➥
                        jndiContext.lookup("jms/javaee7/ConnectionFactory");
      Destination queue = (Destination) jndiContext.lookup("jms/javaee7/Queue");

      try (JMSContext context = connectionFactory.createContext()) {
        context.createConsumer(queue).setMessageListener(new Listener());
      }

    } catch (NamingException e) {
      e.printStackTrace();
    }
  }

  public void onMessage(Message message) {
    System.out.println("Async Message received: " + message.getBody(String.class));
  }
}
```

Message-Oriented middleware (MOM) Java Messaging Service API

- Filtrado de mensajes
- Al crear un mensaje podemos añadir propiedades

```
context.createTextMessage().setIntProperty("orderAmount", 1530);
context.createTextMessage().setJMSPriority(5);
```

- Podemos filtrar los mensajes en función de sus propiedades

```
context.createConsumer(queue, "JMSPriority < 6").receive();
context.createConsumer(queue, "JMSPriority < 6 AND orderAmount < 200").receive();
context.createConsumer(queue, "orderAmount BETWEEN 1000 AND 2000").receive();
```

Message-Oriented middleware (MOM) Message-Driven Beans

- Un MDB es un consumidor asíncrono que es activado por el contenedor como resultado de la llegada de un mensaje.
- Desde el punto de vista del productor se trata de un consumidor normal.
- MBD es parte de la especificación EJB y se ejecutan dentro del contenedor de EJB.

Message-Oriented middleware (MOM) Message-Driven Beans

- Para crear un MDB anotamos la clase con **@MessageDriven**.
- Debe implementar el interfaz **MessageListener**.
- Debe ser una clase pública no final ni abstracta.
- Debe tener constructor vacío
- No debe definir el método **finalize**

```
@MessageDriven(mappedName = "jms/javaee7/Topic")
public class BillingMDB implements MessageListener {

  public void onMessage(Message message) {
    System.out.println("Message received: " + message.getBody(String.class));
  }
}
```

Message-Oriented middleware (MOM) Message-Driven Beans

- Como cualquier EJB podemos inyectar dependencias en él, por ejemplo la persistencia

```
@PersistenceContext
private EntityManager em;
@Inject
private InvoiceBean invoice;
@Resource(lookup = "jms/javaee7/ConnectionFactory")
private ConnectionFactory connectionFactory;
```

Message-Oriented middleware (MOM) Message-Driven Beans

- Un MDB puede también actuar como productor. Recibe un mensaje, lo procesa y lo envía a otro destino.

```
public class BillingMDB implements MessageListener {

  @Inject
  @JMSConnectionFactory("jms/javaee7/ConnectionFactory")
  @JMSSessionMode(JMSContext.AUTO_ACKNOWLEDGE)
  private JMSContext context;
  @Resource(lookup = "jms/javaee7/Queue")
  private Destination printingQueue;

  public void onMessage(Message message) {
    System.out.println("Message received: " + message.getBody(String.class));
    sendPrintingMessage();
  }

  private void sendPrintingMessage() throws JMSException {
    context.createProducer().send(printingQueue, "Message has been received and resent");
  }
}
```

WebSocket

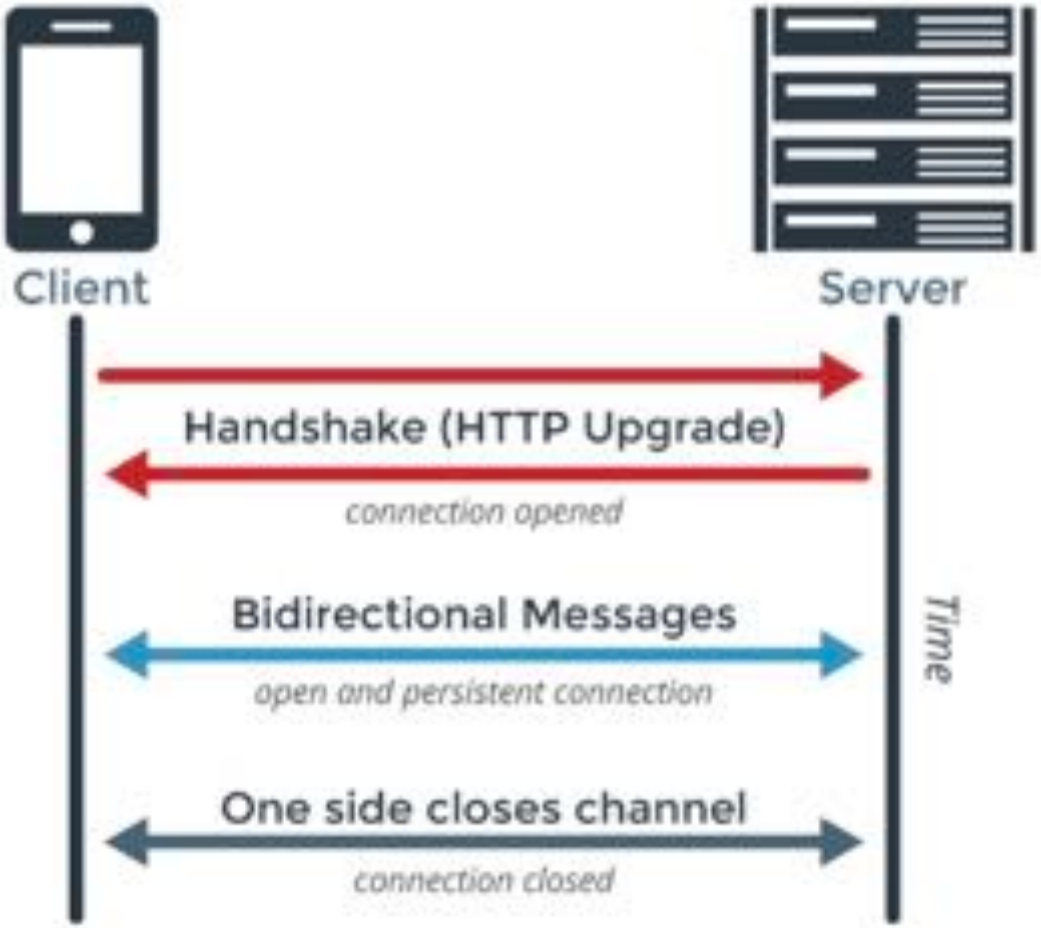
WebSocket
Motivación

- A partir del año 2005, **AJAX**: El servidor no puede iniciar una comunicación con el cliente, hasta que este (cliente) no tome la iniciativa.
- **WebSocket**: Que permiten a los servidores enviar información al cliente en cualquier momento; normalmente cuando tienen nueva información disponible.
- **WebSocket** plantea un modelo sencillo de comunicaciones para la Web que no rompe con las tecnologías ya existentes.

WebSocket
¿Qué es?

- Protocolo que permite crear una comunicación bidireccional sobre una conexión TCP.
- La comunicación se realiza mediante el mismo puerto de HTTP.
- WebSocket es un protocolo independiente a TCP y TLS.
- Dos partes: **Negociación y Transferencia de datos.**

WebSocket
Comunicación



WebSocket Negociación

- Para establecer una conexión, el cliente manda una **petición** de negociación y el servidor una **respuesta**.

Petición WebSocket del navegador

```
GET /demo HTTP/1.1
Host: example.com
Connection: Upgrade
Sec-WebSocket-Key2: 12998 5 Y3 1 .P00
Sec-WebSocket-Protocol: sample
Upgrade: WebSocket
Sec-WebSocket-Key1: 4 @1 46546xW%01 1 5
Origin: http://example.com

^n:ds[4U
```

Respuesta WebSocket del servidor

```
HTTP/1.1 101 WebSocket Protocol Handshake
Upgrade: WebSocket
Connection: Upgrade
Sec-WebSocket-Origin: http://example.com
Sec-WebSocket-Location: ws://example.com/demo
Sec-WebSocket-Protocol: sample

8jKS'y:G*Co,Wxa-
```

WebSocket Negociación

- Los 8 bytes con valores numéricos de estos campos son tokens aleatorios que el servidor usa para construir un token de 16 bytes al final de la negociación para confirmar que ha leído correctamente la petición.

WebSocket Negociación

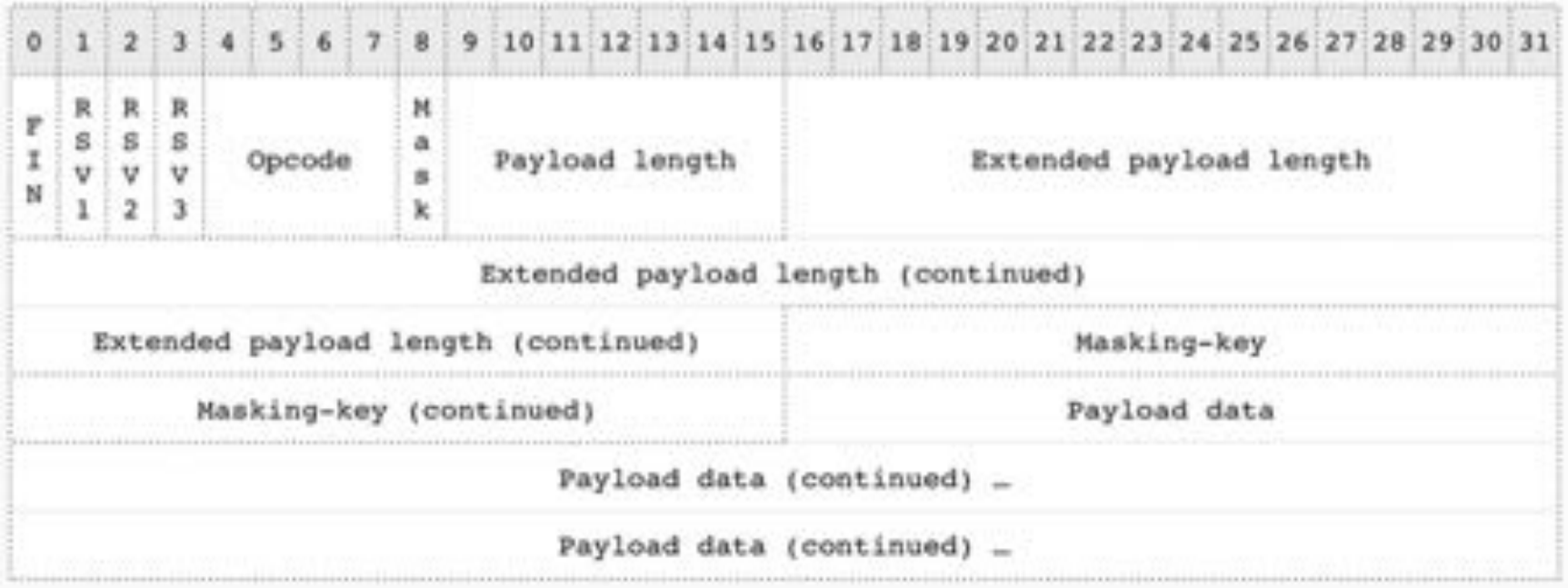
- Tanto en la petición como en la respuesta, se incluyen una serie de cabeceras obligatorias:
  - Obligatorias de HTTP/1.1: **Host**
  - Necesarias para establecer la conexión: **Upgrade, Connection y Sec-WebSocket-***
  - Relacionadas con el modelo de seguridad: **Origin**

WebSocket
Transferencia de datos

- La unidad elemental de transferencia son los mensajes, compuestos por una o más **tramas**.
- Cada **trama** tiene un tipo de datos asociado que coincide con el resto de tramas del mensaje.
- El formato efectivo de la transferencia de datos es binario.

WebSocket Tramas

- La versión más reciente de WebSocket define seis tipos y deja diez más reservados para uso futuro.



WebSocket
Campo de las tramas

- **FIN:** indica que la trama es la última del mensaje.
- **RSV1, RSV2,RSV3:** reservados para su uso por parte de extensiones.
- **Opcode:** define tipo de datos que contiene la trama.
- **Mask:** indica si el campo **Payload data** esta enmascarado. Todas las tramas del cliente lo están.
- **Masking-key:** contiene la máscara utilizada y solo está presente cuando el campo **Mask** es 1 .

WebSocket
Campo de las tramas

- **Payload Length:** define la longitud del campo **Payload data** en bytes. El protocolo establece un mecanismo para usar el campo **Extended payload length** para indicar longitudes mayores de 127 bytes.
- **Payload data:** contiene la información del mensaje.

WebSocket
Cierre

- Es posible cerrar la conexión mediante una negociación de cierre.
- Se inicia enviando un mensaje de control específico, al cual el otro extremo responde con otro mensaje de control para confirmar.
- La negociación de cierre esta pensada para ir acompañada del cierre de la conexión TCP.

WebSocket JavaScript

- Crear objeto WebSocket
- Debemos indicar la URL
- En vez de http -> ws; https -> wss

```
var testSocket = new WebSocket("ws://localhost:8080/websocket/test/");
```

- Puede llevar subprotocolos opcionales

```
var testSocket = new WebSocket("ws://localhost:8080/websocket/test/", "protocolo1");
var testSocket = new WebSocket("ws://localhost:8080/websocket/test/", ["protocolo1","protocolo2"]);
```

WebSocket JavaScript

- Con la conexión abierta podemos enviar información mediante **send**

```
testSocket.send("Esto es un mensaje");
```

- Podemos enviar información compleja mediante JSON

```
var msg = {
    type: "mensaje",
    text: "Esto es un mensaje",
    id: clientID,
    date: Date.now()
}
testSocket.send(JSON.stringify(msg));
```

WebSocket JavaScript

- Para recibir información utilizamos manejadores

```
testSocket.onmessage = function (event) {
    console.log("Recibido:" + event.data);
}
```

- Podemos recibir información compleja mediante JSON

```
testSocket.onmessage = function (event) {
    console.log("Recibido JSON:" + JSON.parse(event.data));
}
```

WebSocket JavaScript

- Disponemos otros manejadores para cuando se conecta y se desconecta el WebSocket

```
testSocket.onopen = function (event) {
        console.log("Conexión abierta");
};

testSocket.onclose = function (event) {
        alert("Conexión cerrada desde el servidor");
};
```

- Es posible cerrar la conexión desde el cliente

```
testSocket.close();
```

WebSocket JAVA

- Anotamos con *@ServerEndpoint(path)*
- Definimos un método para atender las peticiones y la anotamos con @OnMessage

```
@ServerEndpoint("/test/")
public class TestSocket {

    @OnMessage
    public void onMessage(String message, final Session session) {
        try {
            session.getBasicRemote().sendText("Has enviado: "+message);
        } catch (IOException ex) {
            Logger.getLogger(TestSocket.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}
```

WebSocket JAVA

- Podemos controlar la creación y destrucción mediante @OnOpen y @OnClose

```
List<Session> socketSessions = Collections.synchronizedList(new ArrayList<Session>());

@OnOpen
public void open(Session session) {
  socketSessions.add(session);
}

@OnClose
public void close(Session session) {
  socketSessions.remove(session);
}
```

Para ampliar conocimientos

- Antonio Goncalves, Beginning Java EE 7 (caps. 13)
- Especificación de JMS 2.1 (JSR 368): https://jcp.org/en/jsr/detail?id=368
- Especificación de Java API for WebSocket (JSR 356): https://jcp.org/en/jsr/detail?id=356