

Arquitectura Software

Publicación-Suscripción (*Publish-Subscribe*)

Contenidos

- Limitaciones RPC y Broker de objetos
- Middleware orientado a mensajes (MOM)
- Brokers de mensajes
- Patrón Observer
- Publish/Subscribe
- Ejemplos

Brokers de objetos

- Dos limitaciones principales
 - Middleware intra-empresarial
 - Difícil/imposible la interoperabilidad entre Brokers
 - ¿Cómo abrir los sistemas a los clientes?
 - Solución → *Tecnologías Web*
 - Fundamentalmente síncrono →
 - Solución → *Middleware* orientado a mensajes (MOM)

MOM

- Orígenes
 - RPC y brokers de objetos incorporaron versiones asíncronas → quizá poco flexibles
 - Los monitores transaccionales incorporaron interacciones asíncronas → colas de mensajes

MOM ...

- Interoperabilidad basada en mensajes
 - Mensaje, conjunto de datos estructurado, caracterizado por un tipo y unos parámetros
 - conjunto de pares <nombre,valor>
 - XML suele ser la referencia

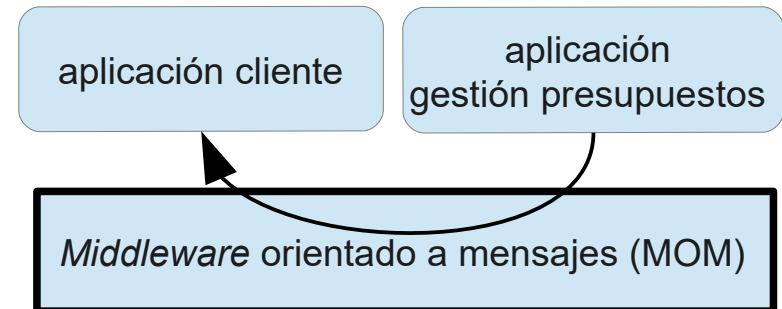
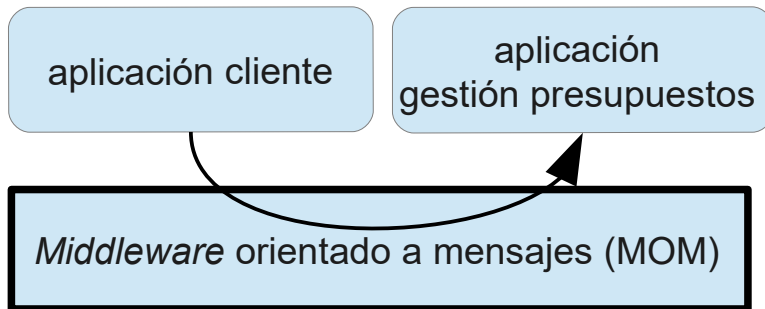
```
Message solicitudPresupuesto{  
    NumReferencia: Integer  
    Cliente: String  
    Artículo: String  
    Cantidad: Integer  
    FechaEntregaSolicitada: Timestamp  
    DirecciónEntrega: String  
}
```

MOM ...

- Interoperabilidad basada en mensajes ...
 - Cliente y proveedor de servicios acuerdan un conjunto de tipos de mensajes

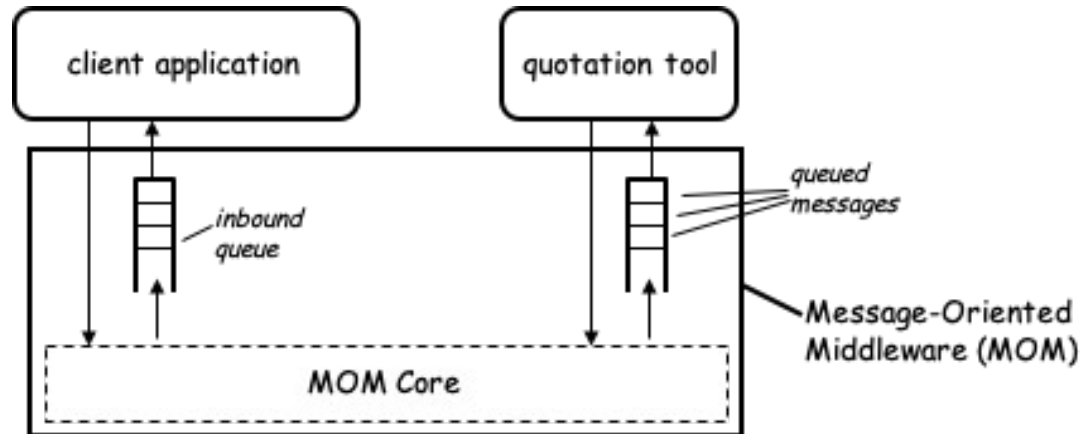
```
Message solicitudPresupuesto {  
  NumReferencia: 325  
  Cliente: ACME, INC  
  Artículo: #115 (bolígrafo azul)  
  Cantidad: 1200  
  FechaEntregaSolicitada: Mayo 15, 2019  
  DirecciónEntrega: C/María Luna, Zaragoza, Spain  
}
```

```
Message presupuesto {  
  NumReferencia: 325  
  FechaEntregaEsperada: Mayo 20,  
  2019  
  Precio: 1200€  
}
```



MOM ...

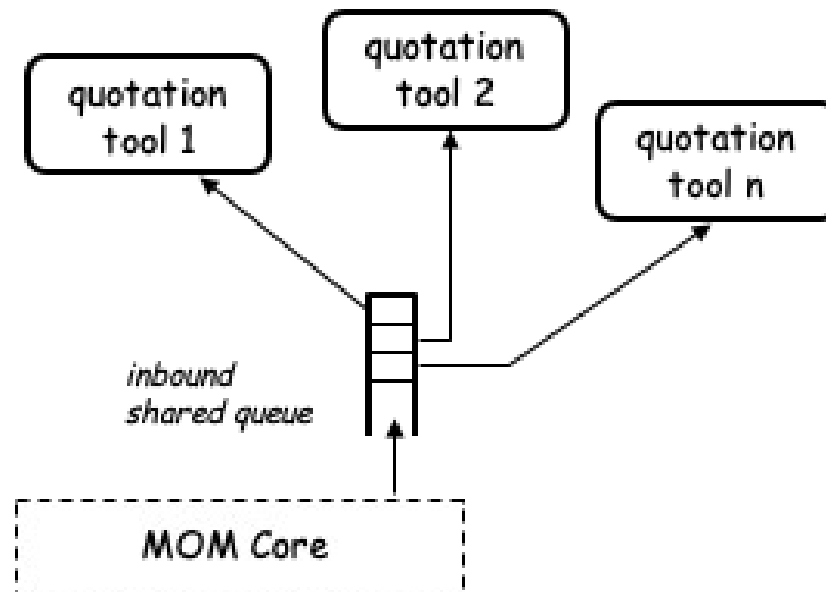
- Interoperabilidad basada en mensajes ...
 - Los mensajes se encolan
 - Cuando el receptor “puede procesar” el mensaje usa las primitivas del sistema MOM → API del MOM
 - Ventajas
 - El receptor tiene el control de cuándo procesa el mensaje
 - No tiene que estar “escuchando”
 - Más robusto (si se cae el receptor ... no pasa nada)
 - Los mensajes tienen fecha de caducidad ... se descartan pasada la misma



MOM ...

- Ventajas ...

- Las colas se pueden compartir entre aplicaciones que proporcionan el mismo servicio o para hacer balanceo de carga (mejorar prestaciones)
- Prioridades en los mensajes



MOM ...

- Ventajas ... más soporte a errores y fallos
 - MOM nos asegura que enviado el mensaje este será recibido *una y sólo una* vez por el destinatario. Incluso si MOM se cae
 - Solución: guardar los mensajes en almacenamiento persistente. Recuperarlos cuando se reinicie MOM
- Ventajas ... colas transaccionales
 - Transacción: conjunto de mensajes que se ejecutan con la propiedad todo o nada
 - Si un mensaje falla se hace *roll-back* “especial”: Quiere decir que se vuelve a poner el mensaje en la cola

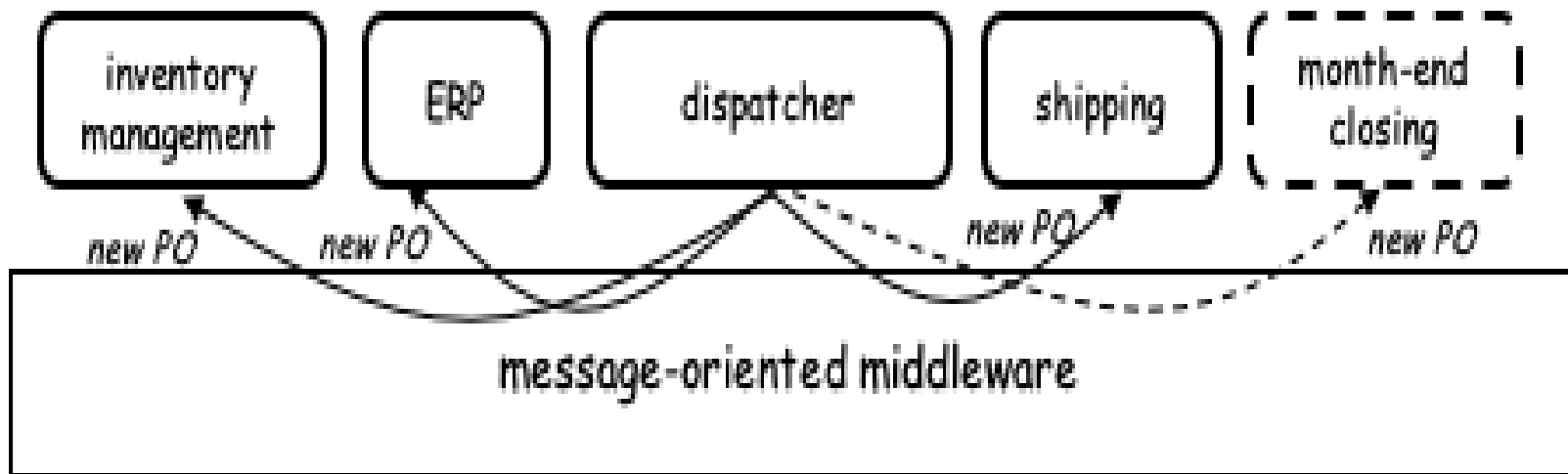
MOM ...

- Limitaciones de MOM

- La unión entre aplicaciones es “punto a punto”
→ Estáticas e inflexibles a la hora de seleccionar las colas a las que enviar el mensaje
- No proporcionan lógica de enrutamiento de mensajes
- Muy limitados para soportar la integración de aplicaciones → Legados

MOM ...

- Ejemplo de limitación
 - Una empresa recibe órdenes de compra (PO) de sus clientes
 - Las procesa con varios sistemas (gestión de existencias, envíos, gestión de cobros, ...)



- Es responsabilidad de “dispatcher” enviar copias del mensaje a cada aplicación

MOM ...

- Ejemplo ...
 - Queremos añadir una nueva aplicación ("*month-end closing*")
 - Con MOM tenemos que modificar el código de "*dispatcher*" !
 - Con MOM es el programa que envía el mensaje el responsable de definir los receptores!
- Ejemplo más dinámico ...
 - Aplicación que monitoriza el mercado de acciones e interopera con aplicaciones interesadas en los cambios de precio de las acciones
- Problema: La responsabilidad de definir el receptor está en el emisor del mensaje

Brokers de mensajes

- Los brokers de mensajes superan esta limitación
- Extienden MOM (son sus descendientes)
 - Lógica de enrutamiento en el middleware
 - Procesamiento de los mensajes
- Ventajas del broker de mensajes
 - Gestiona lógica para encaminar y filtrar los mensajes
 - Ofrece un lenguaje para definir esa lógica
 - Incluso es capaz de procesar los mensajes

Brokers de mensajes ...

- Cómo gestionar la lógica
 - El usuario programa la lógica que identifica, para cada mensaje, las colas a las que debe ser entregado
 - Esa lógica son “reglas” que incluyen:
 - Una condición lógica sobre la información del mensaje
 - Una acción que define las colas a las que se entregarán los mensajes que cumplen la condición
 - Las “reglas” puede residir en:
 - El broker (aplica a todos los mensajes)
 - y en las colas (define el tipo de mensajes que esa cola quiere recibir).
 - La lógica de enrutamiento ya No reside en el mensaje!

Brokers de mensajes ...

- Cómo gestionar la lógica ...
 - El emisor no especifica receptores concretos del mensaje. La lógica de enrutado se basa en:
 - Identidad del emisor
 - Tipo del mensaje
 - Contenido del mensaje
 - Es el broker el que identifica los receptores ejecutando las “reglas” definidas por el usuario

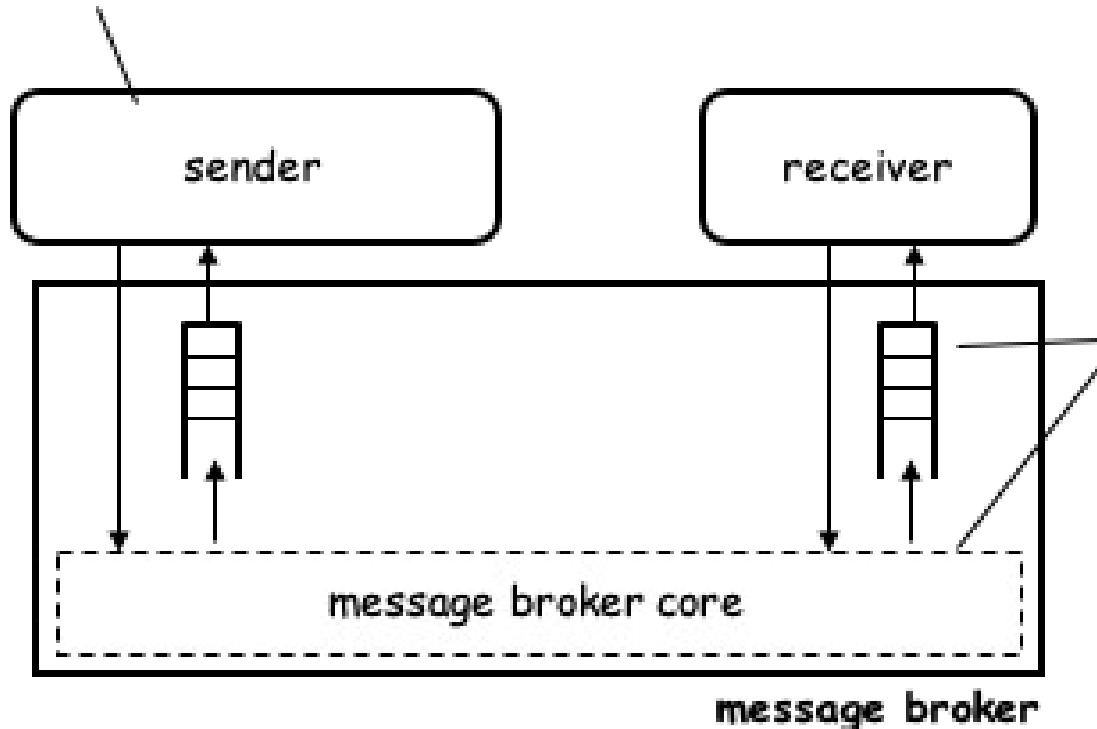
Brokers de mensajes ...

- Ventajas

- Ahora hay un sólo lugar (el middleware) donde modificar cuando la lógica de enrutamiento debe ser actualizada
- *Desacoplan* los programas que envían de los que reciben. Los que envían no dicen a quien; los que reciben no saben de quién
- Además, permiten el procesamiento de los mensajes
 - Por ejemplo, transformaciones para los mensajes
- Pero cuidado con poner mucha lógica y mucho procesamiento en el broker o en las colas → Prestaciones !

Brokers de mensajes ...

in basic MOM it is the sender who specifies the identity of the receivers

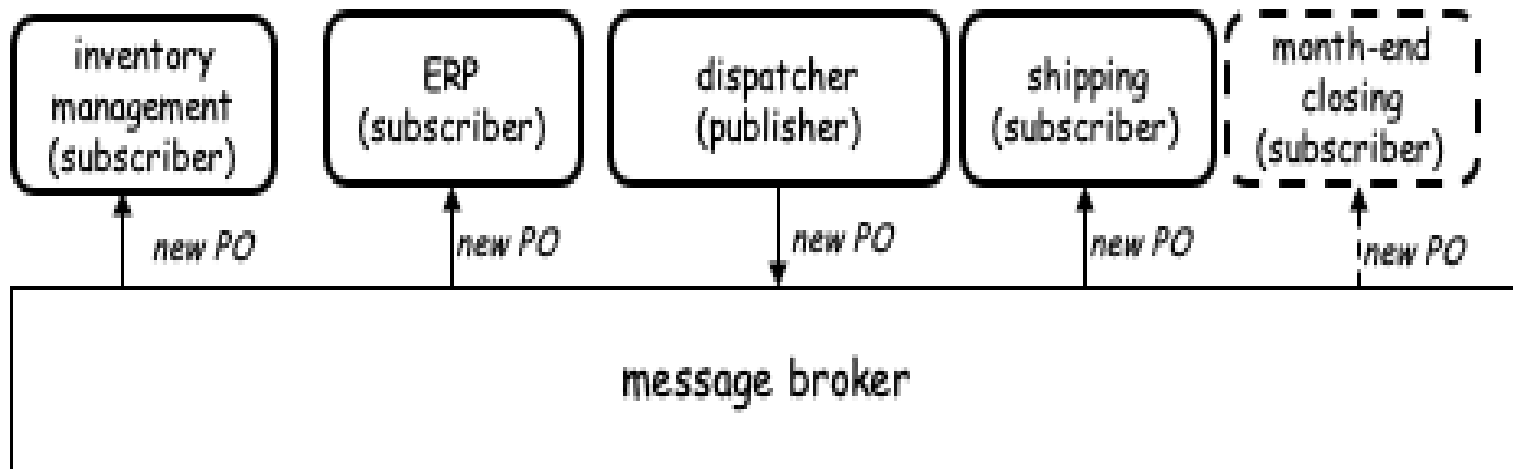


with message brokers, custom message routing logic can be defined at the message broker level or at the queue level

Publish/Subscribe

- Los brokers de mensajes pueden soportar diferentes modelos de interacción
 - P/S es uno de esos modelos
 - En P/S el broker suele llamarse “bus de eventos”
- Características básicas del modelo
 - Por supuesto, basado en mensajes
 - Aplicaciones que publican mensajes en el broker
 - Aplicaciones que consumen mensajes del broker

Publish/Subscribe



- Características básicas del modelo ...
 - Cuando una aplicación envía un mensaje al broker, éste selecciona las aplicaciones suscritas al *tipo* del mensaje y les envía una copia
 - Algunas aplicaciones son a la vez productoras y consumidoras

Publish/Subscribe

- Por otro lado, podemos ver P/S como una variante del archiconocido patrón “Observer”

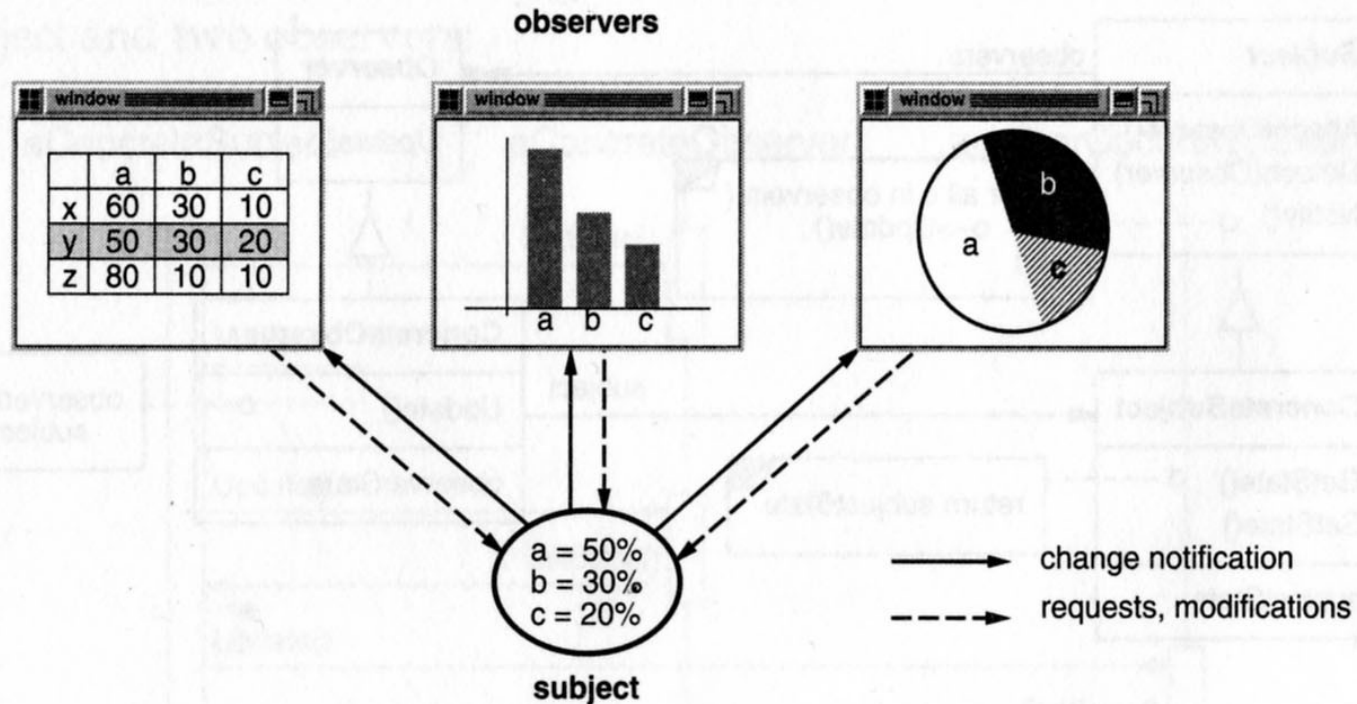
Patrón Observer

- Propósito
 - Definir una *dependencia 1 a muchos* entre objetos de modo que cuando un objeto cambia de estado, todos los que de él dependen son *notificados y actualizados automáticamente*
- Otros nombres
 - *Publish-Subscribe*

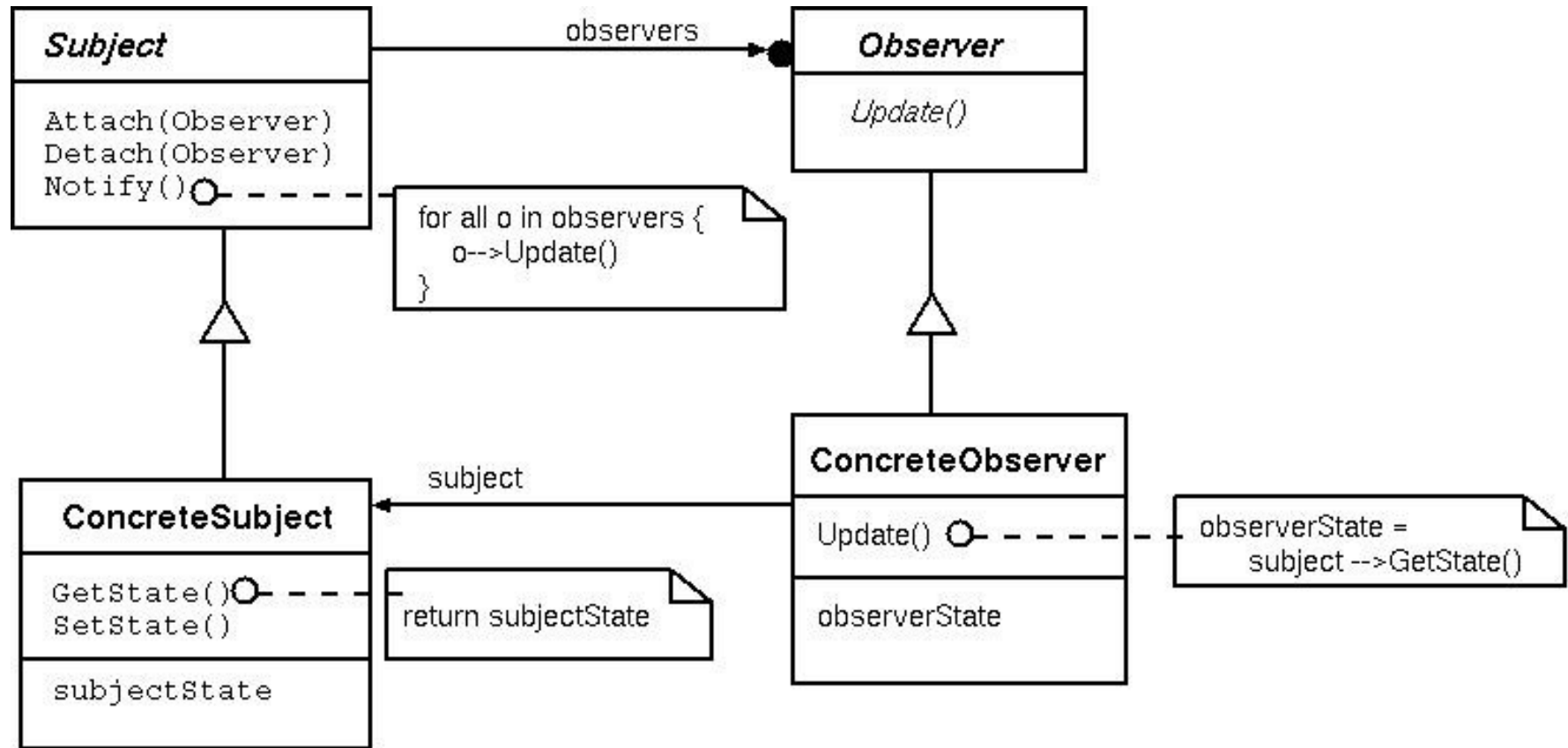
Observer: Motivación

- Efecto lateral de particionar un sistema en clases → necesidad de mantener *consistencia* entre clases relacionadas
- Si hacemos las clases fuertemente acopladas
 - Consistencia +
 - Reusabilidad -

Observer: Motivación ...



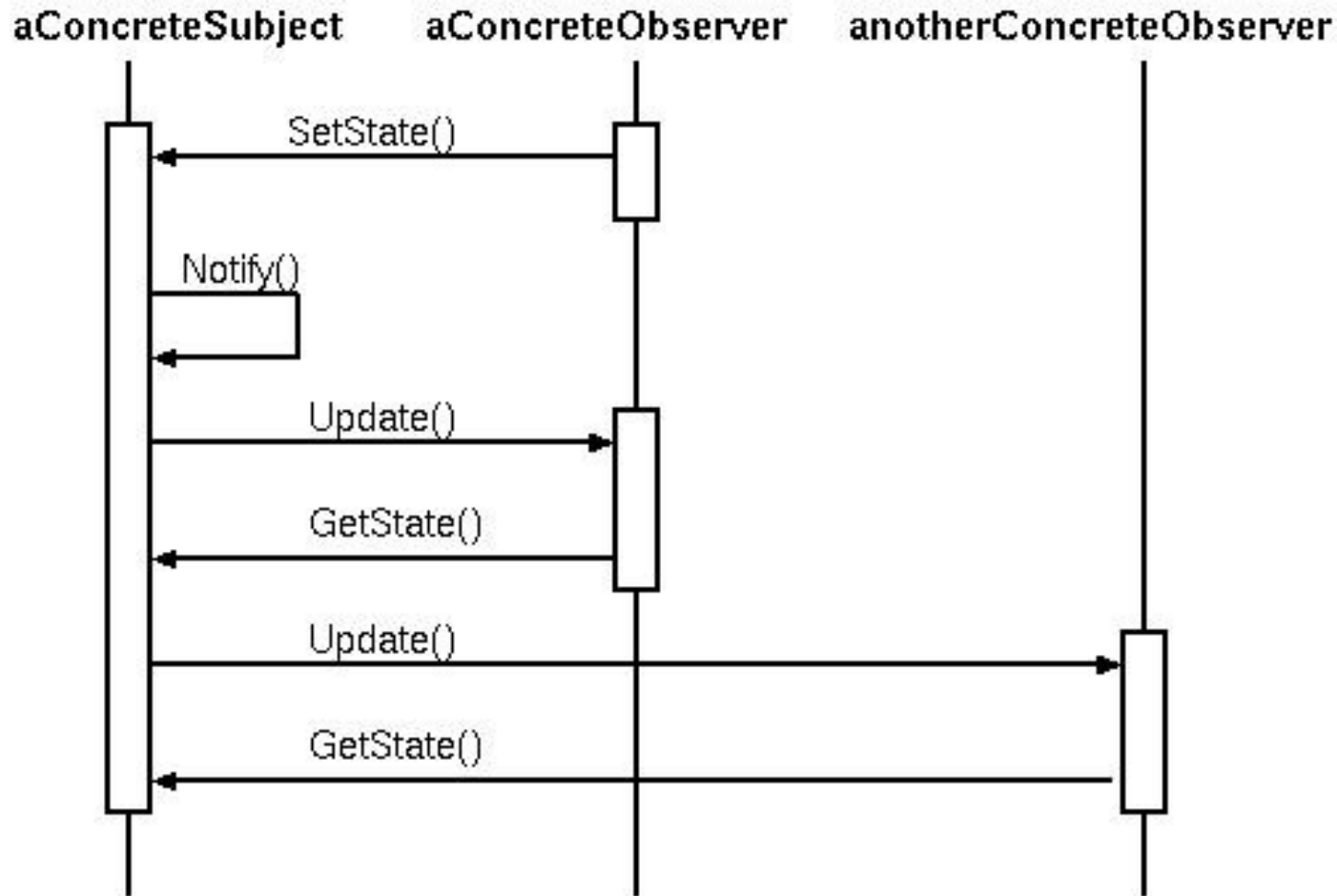
Observer: Estructura



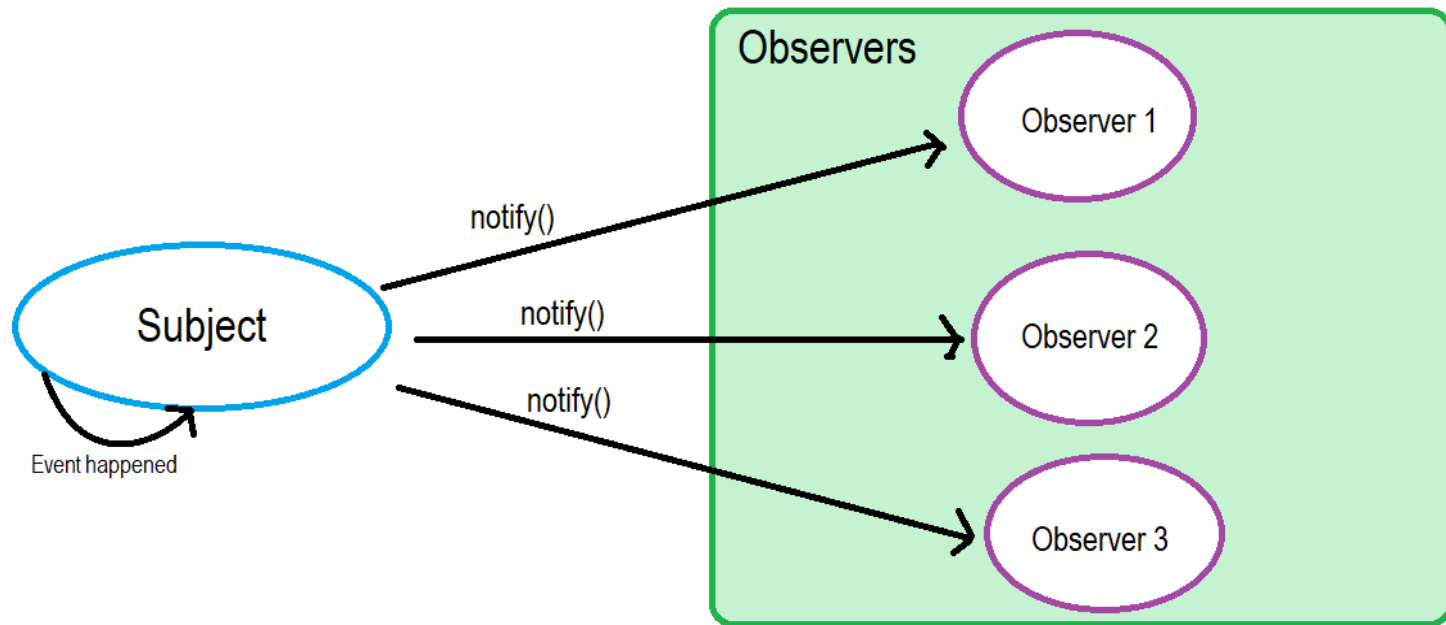
Observer: Participantes

- Subject
 - Conoce a sus observadores
 - Proporciona métodos de suscripción y borrado
- ConcreteSubject
 - Almacena el estado de interés para los observadores concretos
 - Envía notificaciones a sus observadores cuando cambia de estado
- Observer
 - Interfaz para objetos que deben ser notificados
- ConcreteObserver
 - Conoce a su *subject* concreto
 - Almacena estado, consistente con el del *subject*

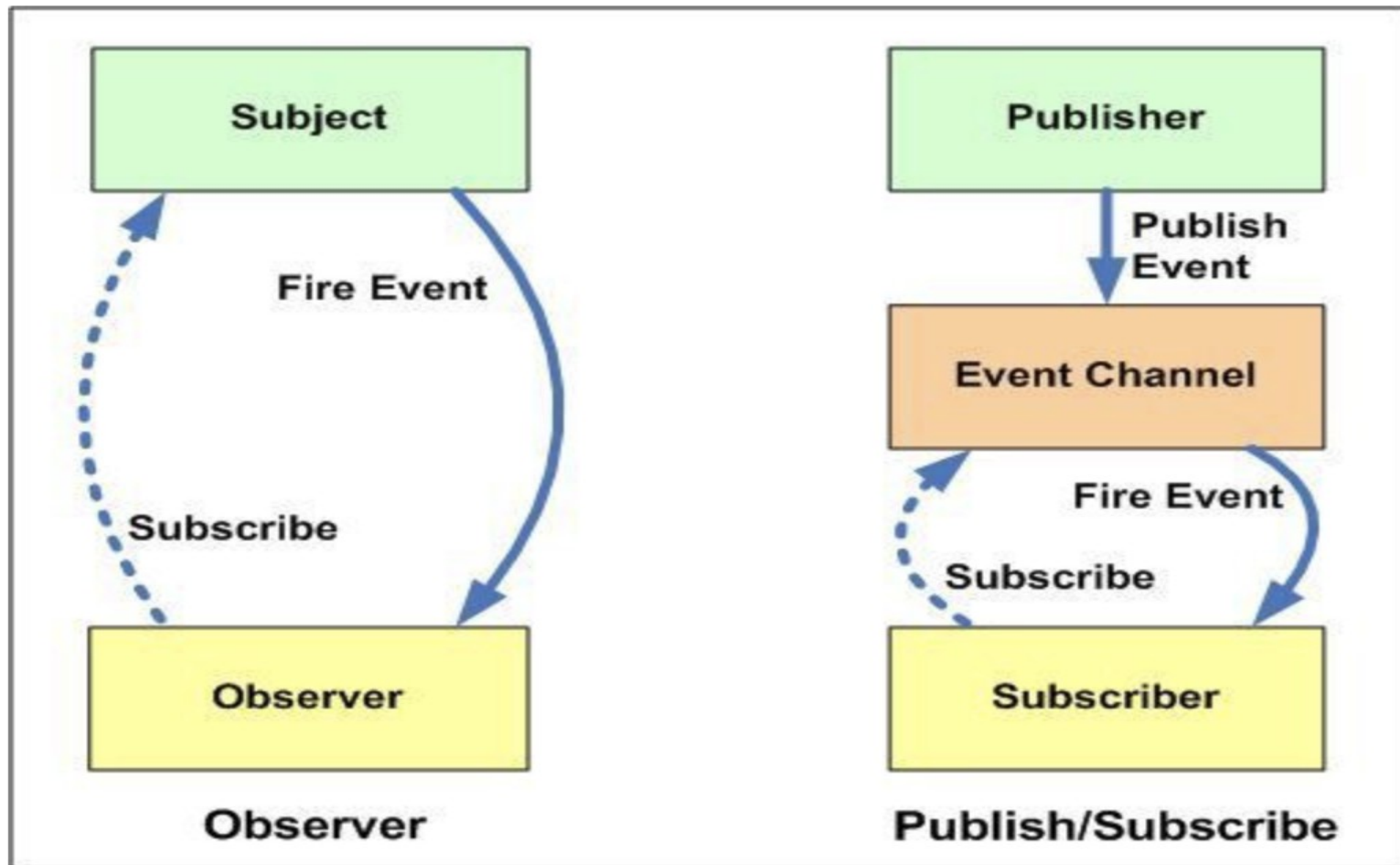
Observer: Colaboraciones



Observer: Colaboraciones



Diferencias Observer y P/S

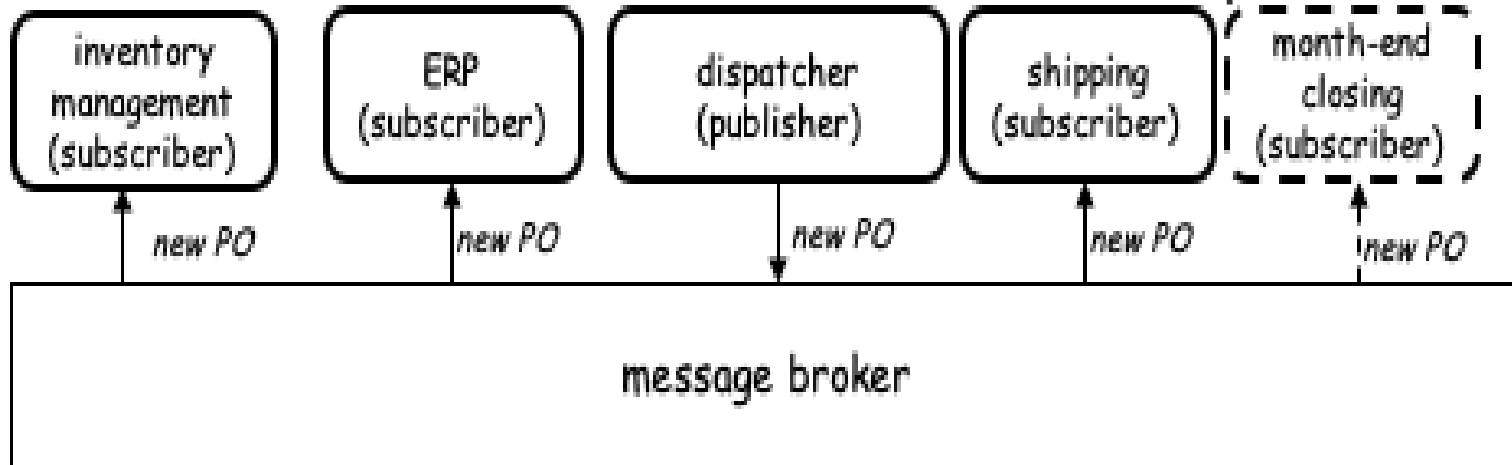


Diferencias Observer y P/S

- Observer se implementa en el mismo espacio de direcciones, P/S es distribuido
- El acoplamiento en P/S es más bajo
 - En Observer, los observadores conocen al `concreteSubject` y el *subject* mantiene una lista de *observers*
- Observer es normalmente síncrono, PS/ es asíncrono
- En Observer la comunicación es *unicast*, en P/S es *multicast*

P/S: Ventajas

- Hay que enviar eventos a N receptores desconocidos
 - P/S **desacopla** productores y consumidores de eventos
- La corrección de los productores no depende de los consumidores, así que se pueden añadir consumidores nuevos sin alterar los productores



P/S: Mensajes

- Hay dos formas de recibir los mensajes
 - Por el tipo del mensaje ("topic")
 - Por ejemplo: tipo newPO
 - Con una condición lógica sobre los valores de los parámetros del mensaje
 - Por ejemplo, "Cliente = 'ACME, INC' and Cantidad > 1500"

P/S: Variantes

- Modelo *push*
 - El publicador envía todos los datos que han cambiado a los suscriptores con cada evento
- Modelo *pull*
 - El publicador notifica que ha habido cambios, pero son los suscriptores los que tienen que averiguar qué ha cambiado
- El *pull* es más flexible, pero exige mayor intercambio de mensajes entre publicadores y suscriptores
- Los mensajes del modelo *push* son menos, pero suelen ser más pesados

P/S: Arquitectura

- Hay un tipo de conector, el conector de publicación-suscripción, que actúa como bus de eventos entre los componentes que publican y los que se suscriben
- Los componentes tendrán puertos de publicación y/o suscripción según vayan a publicar y/o a suscribirse
- Los componentes ponen mensajes en el bus de eventos
- El conector de publicación-suscripción entrega esos mensajes a los componentes que han registrado interés en ellos
- Permite modelar sistemas de procesos u objetos independientes que
 - Reaccionan a eventos generados por su entorno
 - Causan reacciones en otros procesos u objetos mediante los eventos que generan

P/S: Arquitectura ...

- Conector de publicación-suscripción
 - Propiedades: ¿puede un suscriptor encolar nuevos eventos/mensajes mientras procesa uno? ¿Los eventos tienen prioridades? ¿Se fuerza el orden temporal o causal? ¿Es confiable el reparto de eventos?
- Componentes con puertos de publicación y/o suscripción
 - Un publicador tendrá al menos un puerto de publicación y un suscriptor tendrá al menos uno de suscripción
 - Propiedades: qué tipos de mensajes hay, cuándo se bloquea a ciertos anunciantes, si se pueden cambiar las relaciones de suscripción, o añadirse tipos de mensajes o nuevos publicadores en tiempo de ejecución

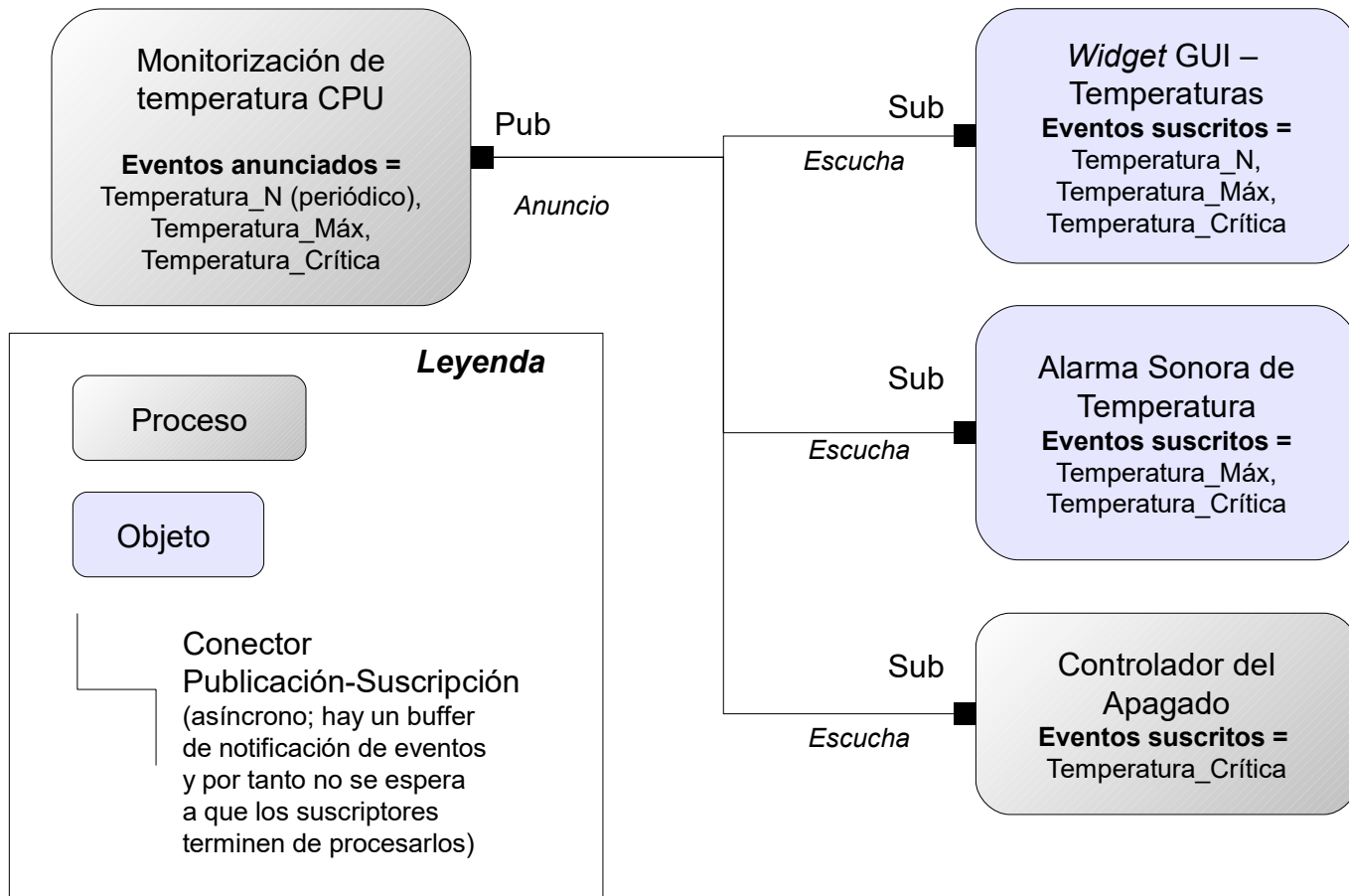
Ejemplos de uso

- GUI, donde las acciones de entrada de bajo nivel del usuario (clicks, pulsación de teclas) se tratan como eventos que se enrutan a los manejadores correspondientes
- Aplicaciones con el patrón Modelo-Vista-Controlador
- Aplicaciones extensibles mediante plugins, que se coordinan por eventos
- Suscripciones a listas de correo, RSS o similares, con filtros por tema
- Redes sociales, donde los cambios en un perfil se comunican a los perfiles “amigos”

Ejemplo – Monitorización de temperatura de CPU

- Tenemos un proceso que monitoriza constantemente el sensor de temperatura de la CPU de nuestro PC y anuncia sus valores
- Otro que cuando se detecta una temperatura crítica inicia el apagado en 30 segundos
- También tenemos un *widget* en el escritorio que muestra una gráfica con la temperatura y que genera una alarma sonora cuando se alcanza la máxima (que es inferior a la crítica) y otra cuando se alcanza la crítica

Ejemplo – Monitorización de temperatura de CPU



Problema – Monitorización de temperatura en Java

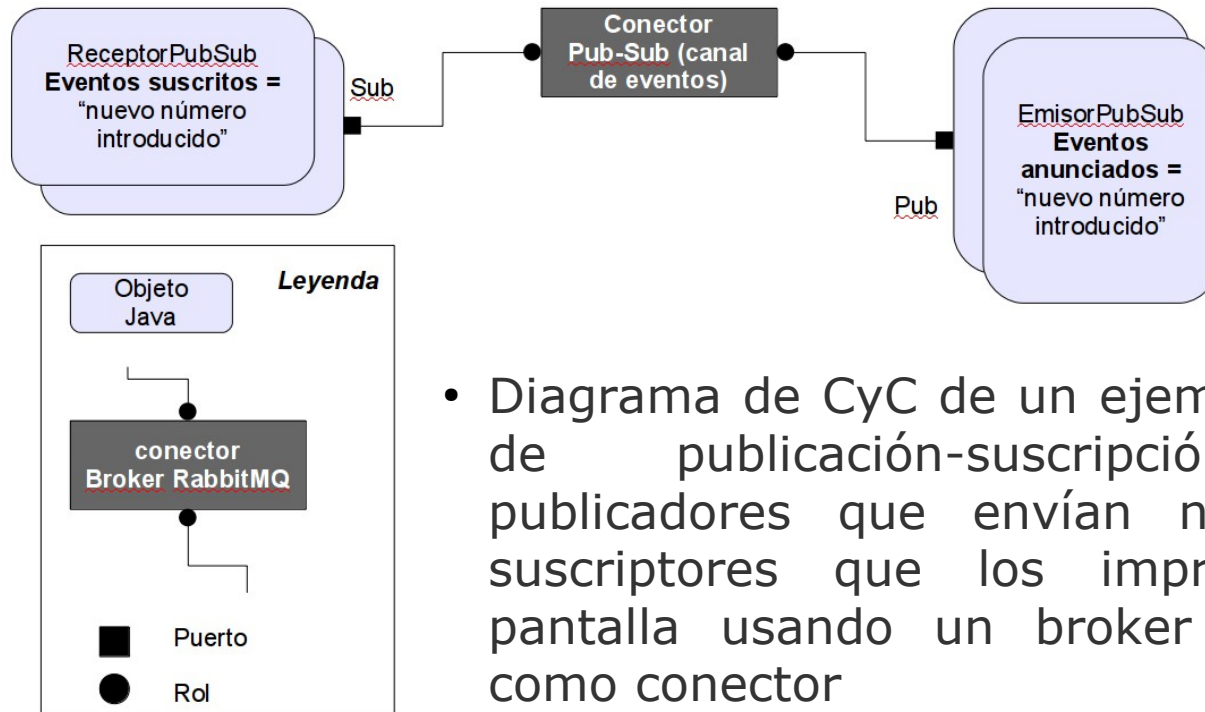
- El ejemplo anterior va a ser implementado en Java
- Asume que serán tres objetos: el *widget* de temperatura, el controlador de apagado y el monitor de temperatura (olvida la alarma sonora)
- Crea una vista de módulos (estilo de generalización) basada en Observable y Observer
- Crea un diagrama de secuencia que documente cómo interaccionan los objetos. Asume que interacción es **síncrona**



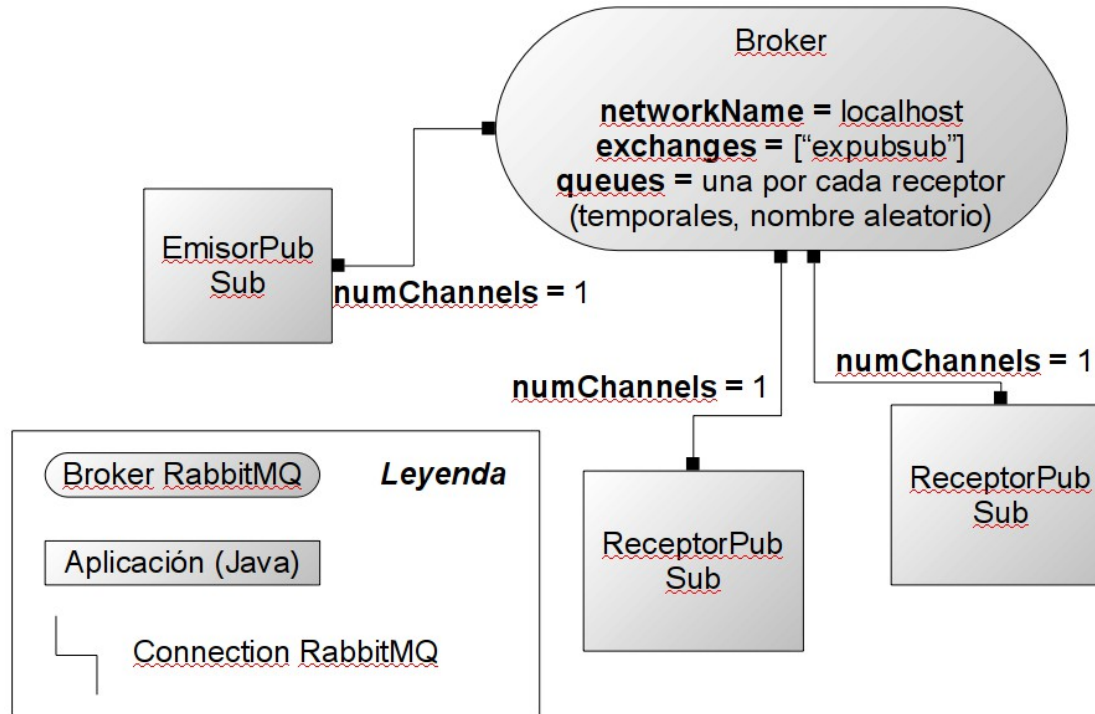
Publicación-suscripción en RabbitMQ

- Un broker de mensajería como RabbitMQ es un candidato claro para implementar conectores asíncronos de publicación-suscripción
- Hemos visto que en RabbitMQ los productores no envían mensajes a colas, los envían a centralitas
 - El tipo de centralita (*exchange type*) determina qué se hace exactamente con los mensajes
- El tipo de centralita *fanout* (de dispersión) sirve para enrutar mensajes a todas las colas que están unidas con ella
 - Si N colas están unidas a una centralita de fanout, cada mensaje que se publica en esa centralita es entregado a todas las N colas
 - Ignoran la clave de enrutado si existe
- Una centralita de *fanout* nos permite implementar el estilo arquitectural de publicación-suscripción con conector asíncrono
- Vamos a ver un ejemplo simple: 1 publicador y N suscriptores que escuchan todos los eventos
 - Si queremos que los suscriptores puedan decidir el tipo de eventos que les interesan, podríamos crear varias centralitas: alguien que quiera recibir eventos relacionados con el tema X, tendría que escuchar en una cola conectada a la centralita X
 - O podríamos usar las centralitas de temas que ofrece RabbitMQ
 - Notad que el tipo de publicación-suscripción que tenemos es el de canal de eventos: podríamos añadir más publicadores sin tener que cambiar nada en los suscriptores

Publicación-suscripción en RabbitMQ



Publicación-suscripción en RabbitMQ



- Otro diagrama de CyC del ejemplo (ilustra un escenario con un emisor y dos receptores). En este caso se enfatiza la configuración de RabbitMQ en lugar de mostrar el estilo publicación-suscripción


Ejemplo Pub-Sub: EmisorPubSub.java

```
// Creamos una conexión al broker RabbitMQ en localhost
ConnectionFactory factory = new ConnectionFactory();
factory.setHost("localhost");
Connection connection = factory.newConnection();
// Con un solo canal
Channel channel = connection.createChannel();

// Declaramos una centralita de tipo fanout llamada EXCHANGE_NAME
channel.exchangeDeclare(EXCHANGE_NAME, "fanout");
```

En lugar de una cola, declaramos una centralita tipo fanout con nombre EXCHANGE_NAME ("expubsub") en el emisor

Ejemplo Pub-Sub: EmisorPubSub.java

```
String message = "Mensaje: " + messageNumber;  
// Publicamos el mensaje en la centralita EXCHANGE_NAME declarada  
// antes. La clave de enrutado la dejamos vacía (la va a ignorar),  
// y no indicamos propiedades para el mensaje (por ejemplo,  
// el mensaje no será durable)   
channel.basicPublish(EXCHANGE_NAME, "", null, message.getBytes());  
System.out.println(" [x] Enviado '" + message + "'");
```

Y publicamos en la centralita con nombre que hemos declarado en lugar de en la centralita por defecto (sin nombre) de los ejemplos vistos en filtro y tubería

Ejemplo Pub-Sub: ReceptorPubSub.java

```
// Declaramos una centralita de tipo fanout llamada EXCHANGE_NAME
channel.exchangeDeclare(EXCHANGE_NAME, "fanout");
// Creamos una nueva cola temporal (no durable, exclusiva y
// que se borrará automáticamente cuando nos desconectemos
// del servidor de RabbitMQ). El servidor le dará un
// nombre aleatorio que guardaremos en queueName
String queueName = channel.queueDeclare().getQueue();
// E indicamos que queremos que la centralita EXCHANGE_NAME
// envíe los mensajes a la cola recién creada. Para ello creamos
// una unión (binding) entre ellas (la clave de enrutado
// la ponemos vacía, porque se va a ignorar)
channel.queueBind(queueName, EXCHANGE_NAME, "");
```

Declaramos la centralita igual que en `EmisorPubSub`. Creamos una cola nueva temporal en cada receptor y la unimos a la centralita

Ejemplo Pub-Sub: ReceptorPubSub.java

```
// El objeto consumer guardará los mensajes que lleguen  
// a la cola queueName hasta que los usemos  
QueueingConsumer consumer = new QueueingConsumer(channel);  
// autoAck a true  
channel.basicConsume(queueName, true, consumer);
```

Y ahora consumimos los mensajes desde la cola temporal cuyo nombre hemos guardado en queueName

Tenéis el ejemplo completo en

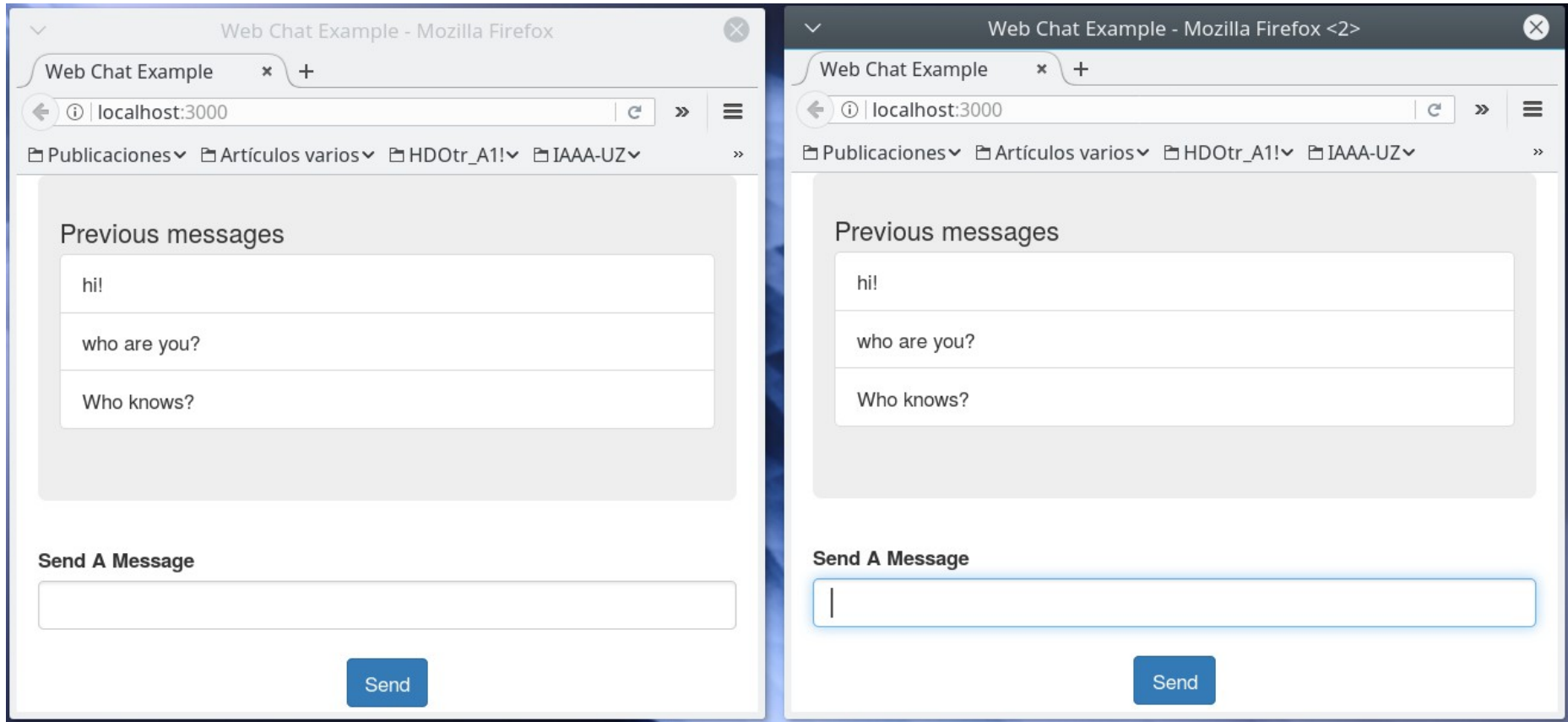
<https://github.com/UNIZAR-30245-ARQS/examples-rabbit>

Ejemplo: Web Chat

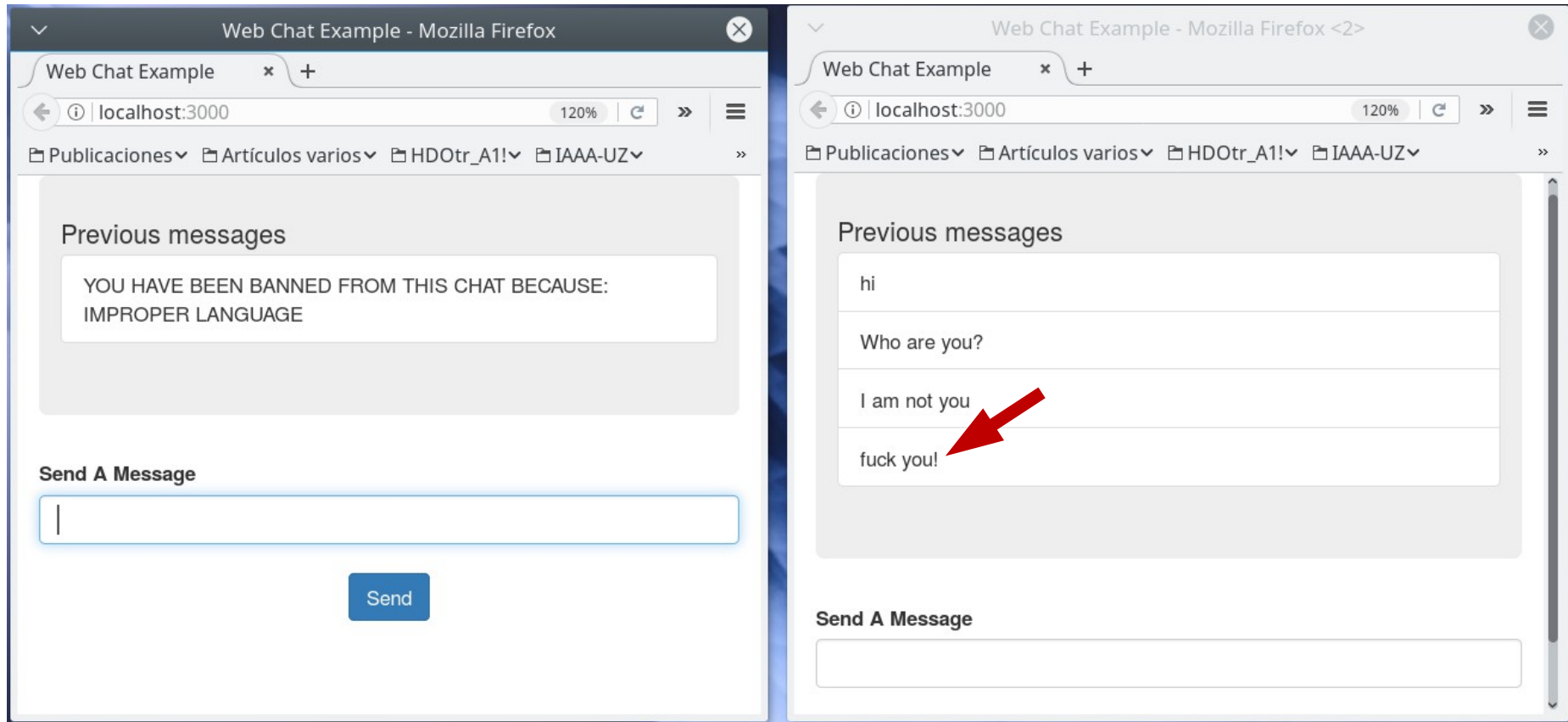
Web Chat

- Queremos diseñar una aplicación web para que N usuarios puedan chatear
 - Funcionalidad muy básica: todos los usuarios ven los mensajes de todos
- También queremos vigilar los mensajes y, si aparecen tacos, desconectar al usuario que los empleó
- Vamos a ver una implementación totalmente distribuida y fácilmente escalable basada en el estilo de publicación-suscripción
 - Node.js con socket.io para gestionar la comunicación asíncrona entre los navegadores, y RabbitMQ para gestionar la comunicación asíncrona con el software que “vigila” los mensajes
 - El código está en <https://github.com/UNIZAR-30245-ARQS/web-chat>

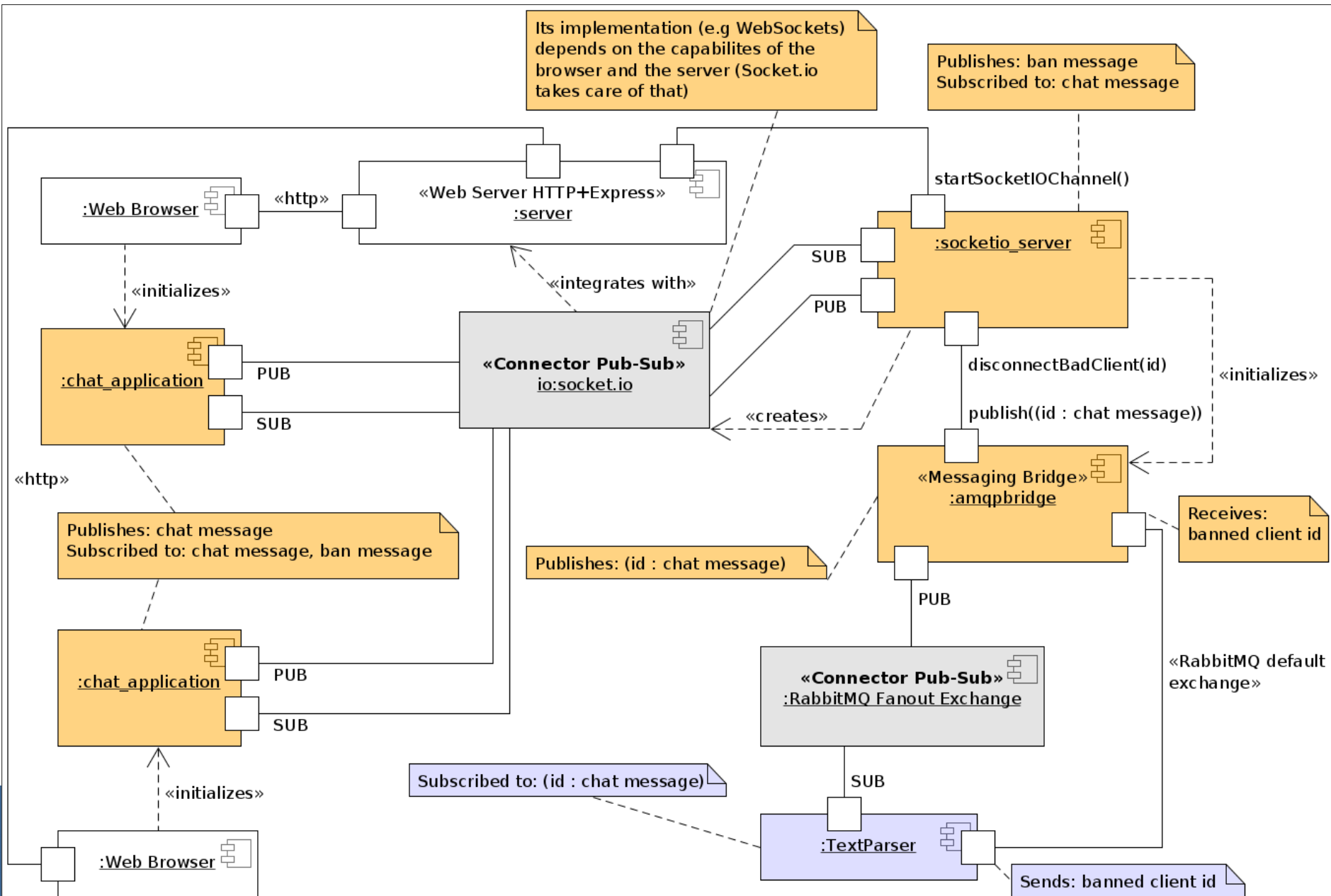
Web Chat



Web Chat



Web Chat: C-y-C



server.js

Lanza un servidor web en el puerto CHAT_HTTP_PORT (lo lee del entorno y si no está usa el 3000).

Al recibir una petición http en la ruta raíz (/) devuelve chat_application.html.

Integra socket.io en el servidor web para que reciba las peticiones que lleguen desde los navegadores.

```
// Return the client (chat_application.html) when a web browser points at the / path
app.get('/', function(req, res){
    res.sendFile(__dirname + '/chat_application.html');
});

// Launch the web server on the port CHAT_HTTP_PORT
http.listen(CHAT_HTTP_PORT, function(){
    console.log('listening on *:', CHAT_HTTP_PORT);
});

socketio_server.startSocketIOChannel(http);
```

TextParser.java

Declara una centralita de tipo fanout llamada EXCHANGE_NAME donde recibirá mensajes del chat a través de una cola temporal

- Como es de tipo fanout, podríamos añadir al sistema otros parsers similares sin alterar para nada a este

Declara una cola BANNED_QUEUE_NAME donde publicará el id de los clientes de chat “infractores”

Y a partir de ahí, lee mensajes de EXCHANGE_NAME, y si encuentra palabras ofensivas publica el id del infractor en BANNED_QUEUE_NAME

- A través de la centralita por defecto, es decir no usando el estilo de publicación-suscripción, sino una conexión punto a punto de mensajería (asíncrona)
 - Aún así TextParser no necesita saber nada sobre quién va a leer ese mensaje (ni siquiera si lo va a leer alguien), el broker de mensajería siempre ayuda a desacoplar a emisores y receptores

chat_application.html

Al pulsar el botón del formulario, envía a través del conector pub-sub (socket) un mensaje con el contenido de la caja de texto.

Cuando llegue un mensaje (*chat message*), escríbelo.

Cuando llegue un mensaje de expulsión (*ban message*), comprueba si es para tu id, y si lo es desconéctate y escribe el texto asociado al mismo.

```
var socket = io();
$('#form').submit(function(){
    socket.emit('chat message', $('#m').val());
    $('#m').val('');
    return false;
});
socket.on('chat message', function(msg){
    $('#messages').append($('- 

```

socketio_server.js

Cuando desde una chat_application se publique un mensaje (*chat message*), envíalo a todos los suscriptores (las otras chat_application)

- También re-envíalo hacia el puente con AQMP añadiendo el id del publicador.

Cuando desde el puente con AMQP llamen a disconnectBadClient(id), envía un mensaje a todos los suscriptores (todas las chat_application) con el id del infractor.

Inicializa el puente con AMQP.

```
function startSocketIOChannel(http) {
  var io = require('socket.io')(http);
  io.on('connection', function(socket){
    console.log('user ' + socket.id + ' connected');
    socket.on('chat message', function(msg){
      console.log('message: ' + msg);
      // Send message to every connected client
      io.emit('chat message', msg);
      // Send it to the messaging middleware (AMQP broker) too so the text parser c
      // We use the client id as a correlation id.
      // Separating them with ":" is not a robust encoding of id+msg, but it is ver
      amqpbridge.publish("", consts.QUEUE_NAME, new Buffer(socket.id + ":" + msg));
    });
  });

  function disconnectBadClient(id) {
    io.emit('ban message', id+':IMPROPER LANGUAGE');
  }

  amqpbridge.setBadClientBehavior(disconnectBadClient);
  // Launch the connection to the AMQP Broker
  amqpbridge.startAMQP();
}
```

amqpbridge.js

Conecta al broker AMQP que está en la URL indicada.

Cuando estés conectado lanza al publicador que enviará los *chat messages*, conforme se los vayamos pasando, a la centralita EXCHANGE_NAME (de tipo *fanout*) en ese broker.

Y lanza al suscriptor que escuchará en la cola BANNED_QUEUE_NAME del broker esperando que el Text Parser envíe un mensaje con el id de algún cliente al que hay que desconectar.

```
function startAMQP() {
  amqp.connect(consts.BROKER_URL + "?heartbeat=60", function(err, conn) {
    if (err) {
      console.error("[AMQP]", err.message);
      return setTimeout(start, 1000);
    }
    conn.on("error", function(err) {
      if (err.message !== "Connection closing") {
        console.error("[AMQP] conn error", err.message);
      }
    });
    conn.on("close", function() {
      console.error("[AMQP] reconnecting");
      return setTimeout(start, 1000);
    });
    console.log("[AMQP] connected");
    amqpConn = conn;
    whenConnected();
  });
}
```

```
// Once connected to the AMQP broker, we launch the publisher and subscriber to that broker
```

```
function whenConnected() {
  startPublisher();
  startSubscriber();
}
```

amqpbridge.js

startPublisher declara la centralita de fanout EXCHANGE_NAME (la crea solo si no existe), luego crea una cola con nombre aleatorio para poder publicar sobre esa centralita (también intenta enviar mensajes que no se pudieron enviar en su momento y se guardaron en offlinePubQueue).

startSubscriber declara la cola BANNED_QUEUE_NAME (la crea solo si no existe) y añade la función processMsg(id) como consumidora de los mensajes que aparezcan en esa cola (que contendrán el id de los infractores según haya determinado TextParser)

- processMsg llamará a notifyBadClientId, que es una función que debe ser proporcionada por el código que usa a este módulo, en nuestro caso socketio_server

publish (exportada por el módulo y a la que se llama desde socketio_server) es la función donde se envían mensajes a través del exchange/routingKey que se le indique

- Y si da algún error, los añade a offlinePubQueue para que se pueda reintentar más tarde

Un posible despliegue

