



Tema 5

Comunicación indirecta

Carlos Montellano



Contenido

1. Comunicación en Grupo

2. Publicador/Suscriptor

3. Cola de Mensajes

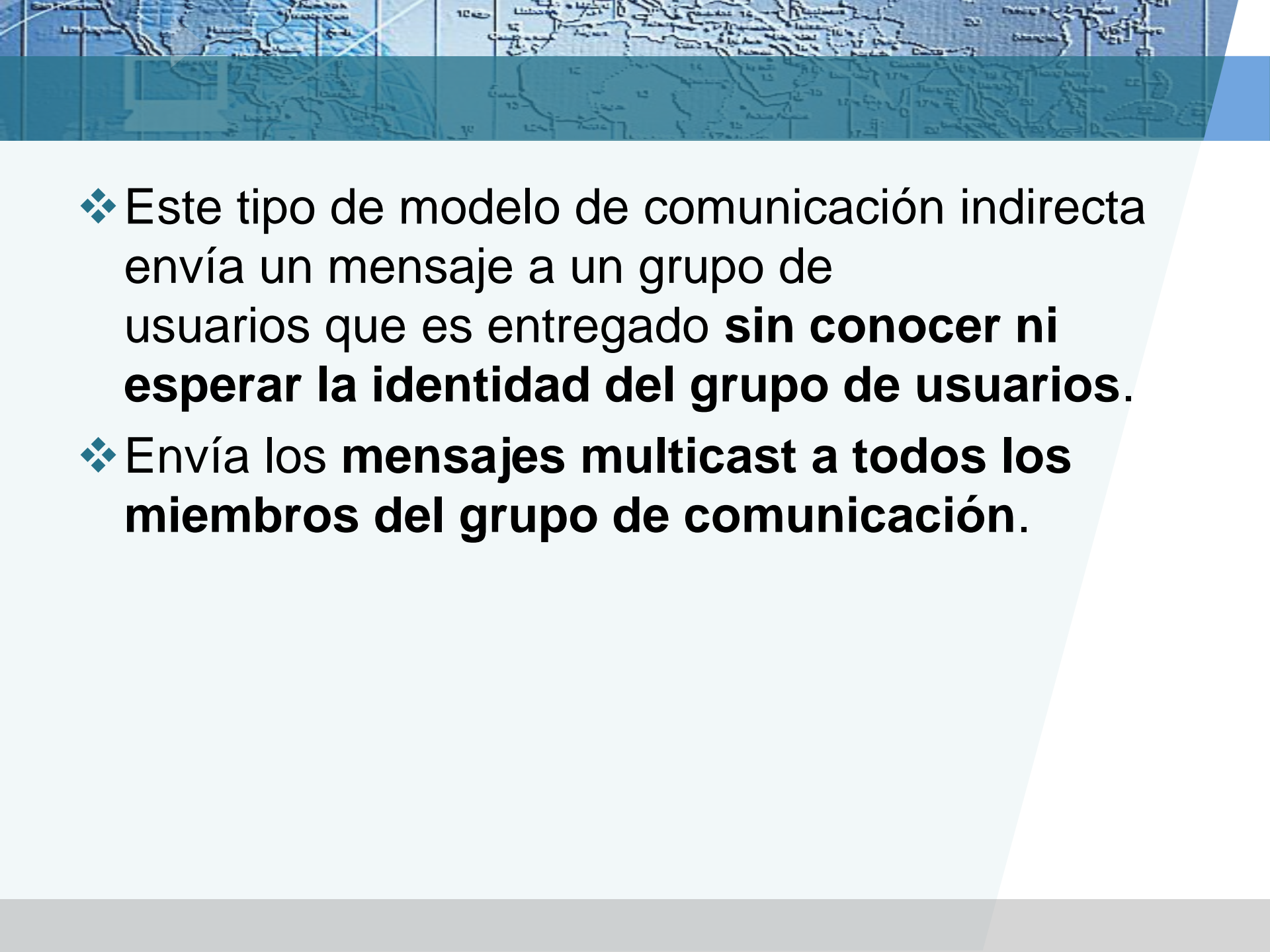
4. Enfoque Memoria Compartida

Espacio y tiempo acoplado en sistemas distribuidos

	Tiempo acoplado	Tiempo desacoplado
Espacio Acoplado	<p>Propiedades:</p> <p>comunicación dirigida hacia un receptor o receptor dado; El receptor debe existir en el momento en el tiempo.</p> <p>Ejemplos: Paso de mensajes, invocación remota</p>	<p>Propiedades: comunicación dirigida hacia un receptor o receptor dado; emisor y receptor pueden tener tiempo de vida independientes</p> <p>Ejemplos: e</p>
Espacio Desacoplado	<p>Propiedades: Emisor no necesita conocer la identidad del receptor; receptor(es) deben existir en ese momento de tiempo</p> <p>Ejemplos: IPmulticast</p>	<p>Propiedades: Emisores no necesitan conocer la identidad de los receptores, emisores y receptores pueden tener tiempo de vida independientes</p> <p>Ejemplos: Paradigmas de comunicación indirecta</p>



Comunicación en Grupo

- 
- A background image showing a map of the Americas, with North and South America visible. The map is in a light blue and white color scheme, with a darker blue overlay on the right side.
- ❖ Este tipo de modelo de comunicación indirecta envía un mensaje a un grupo de usuarios que es entregado **sin conocer ni esperar la identidad del grupo de usuarios.**
 - ❖ Envía los **mensajes multicast a todos los miembros del grupo de comunicación.**

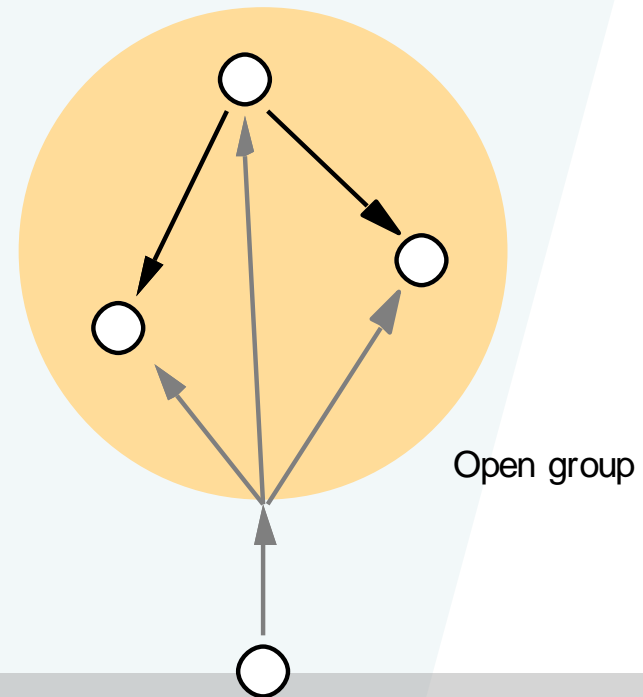
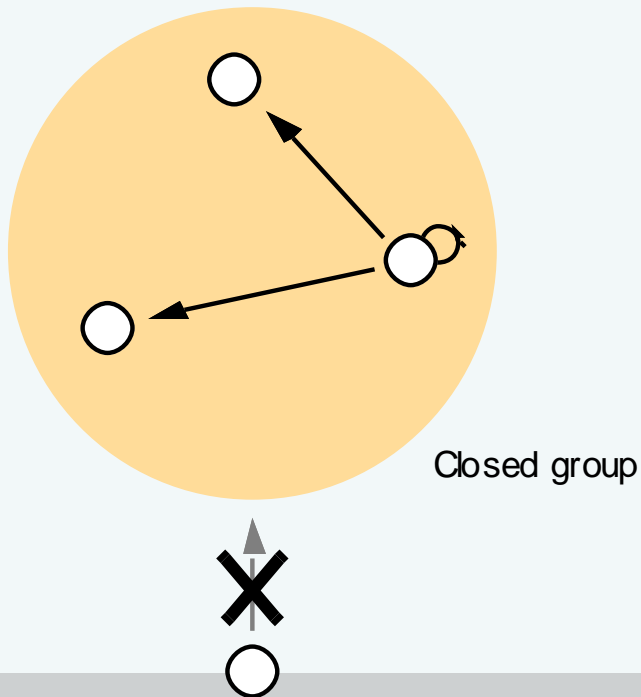
A background map of South America, showing the continent's outline and major cities. The map is in a light blue and white color scheme, with a darker blue overlay at the top where the title is located.

Características

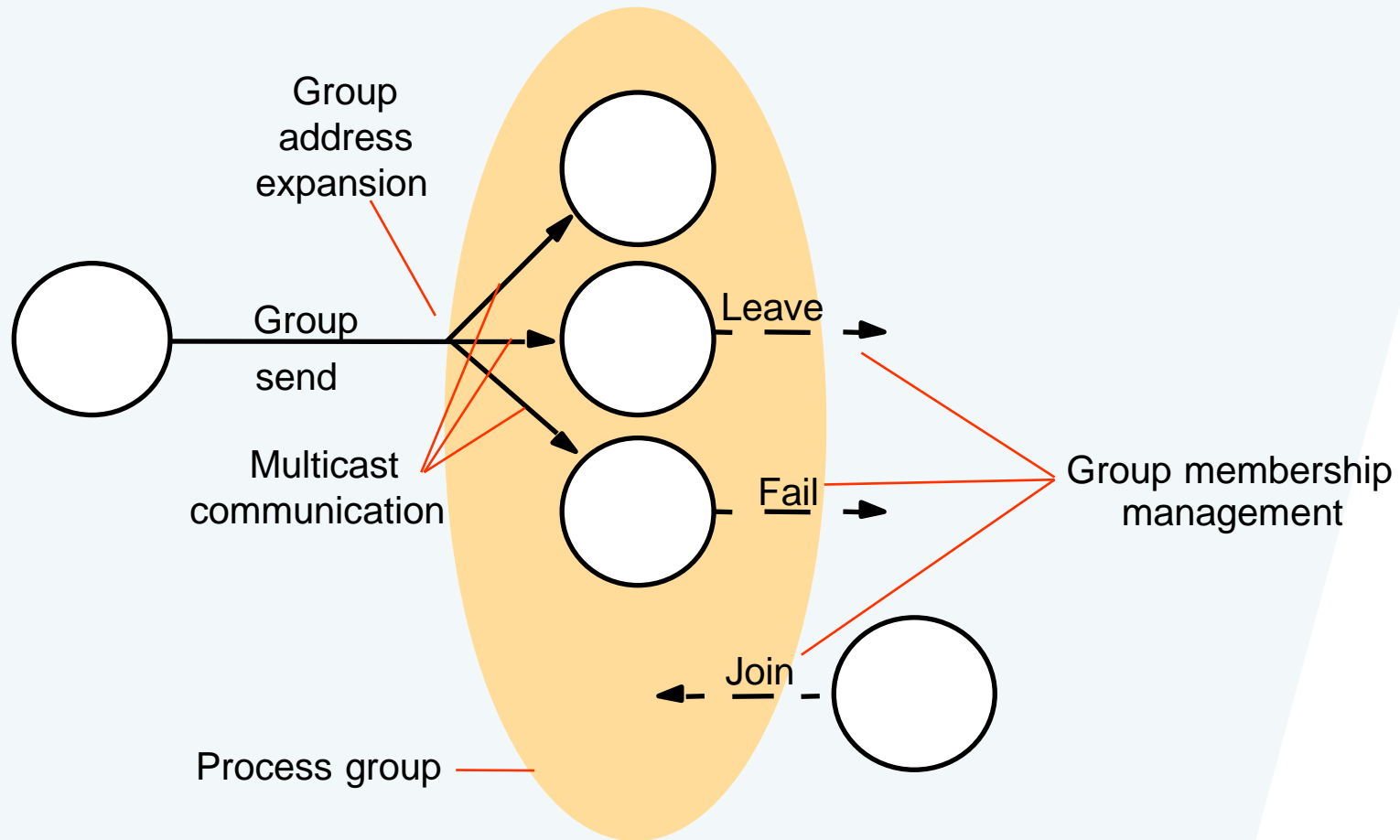
- ❖ Un sistema de **difusión de Información confiable** para que lleguen los mensajes a un gran número de clientes.
- ❖ Soportan aplicaciones de colaboración, como pueden ser los juegos dónde los eventos nuevos deben ser **difundidos a todos los usuarios**, dando una vista común a todos ellos.
- ❖ **Tolerancia a fallos**, en la que también se actualice de manera constante todos los datos replicados.
- ❖ Apoyo a **sistemas de gestión** para poder supervisar el sistema. Por ejemplo sistemas de gestión que ayuden a balancear la carga.

Grupos abiertos y Cerrados

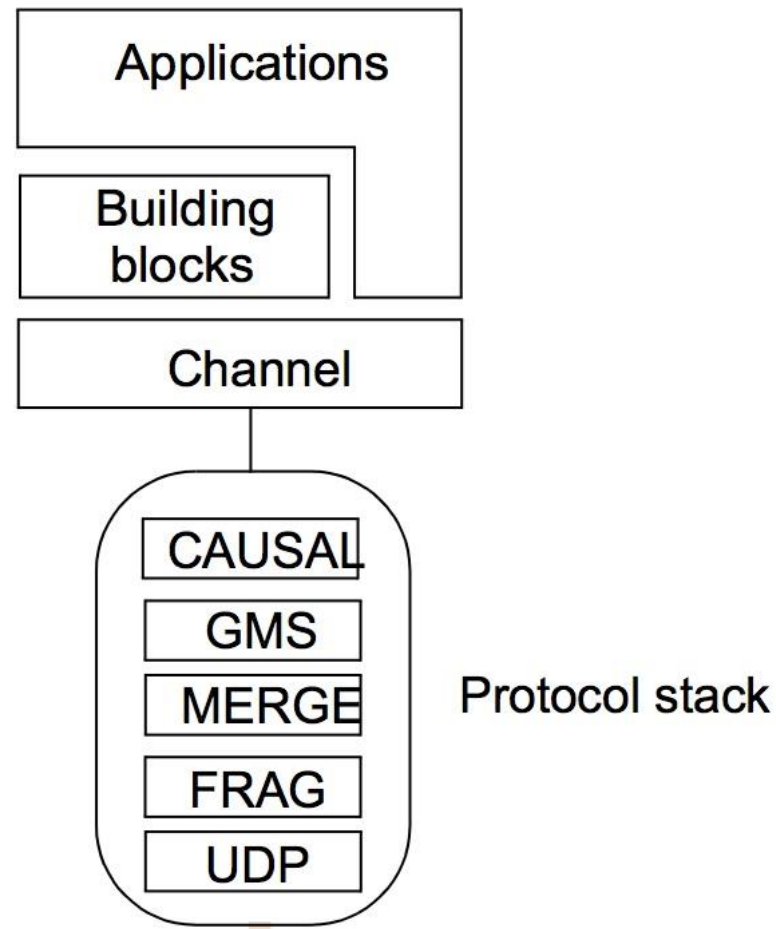
Podemos distinguir grupos de **comunicación abiertos y cerrados**, los primeros pueden recibir mensajes desde otros sitios de fuera del grupo, mientras que los segundos no pueden recibir mensajes de otros miembros que no sean del grupo.



El papel de la gestión de miembros de grupo



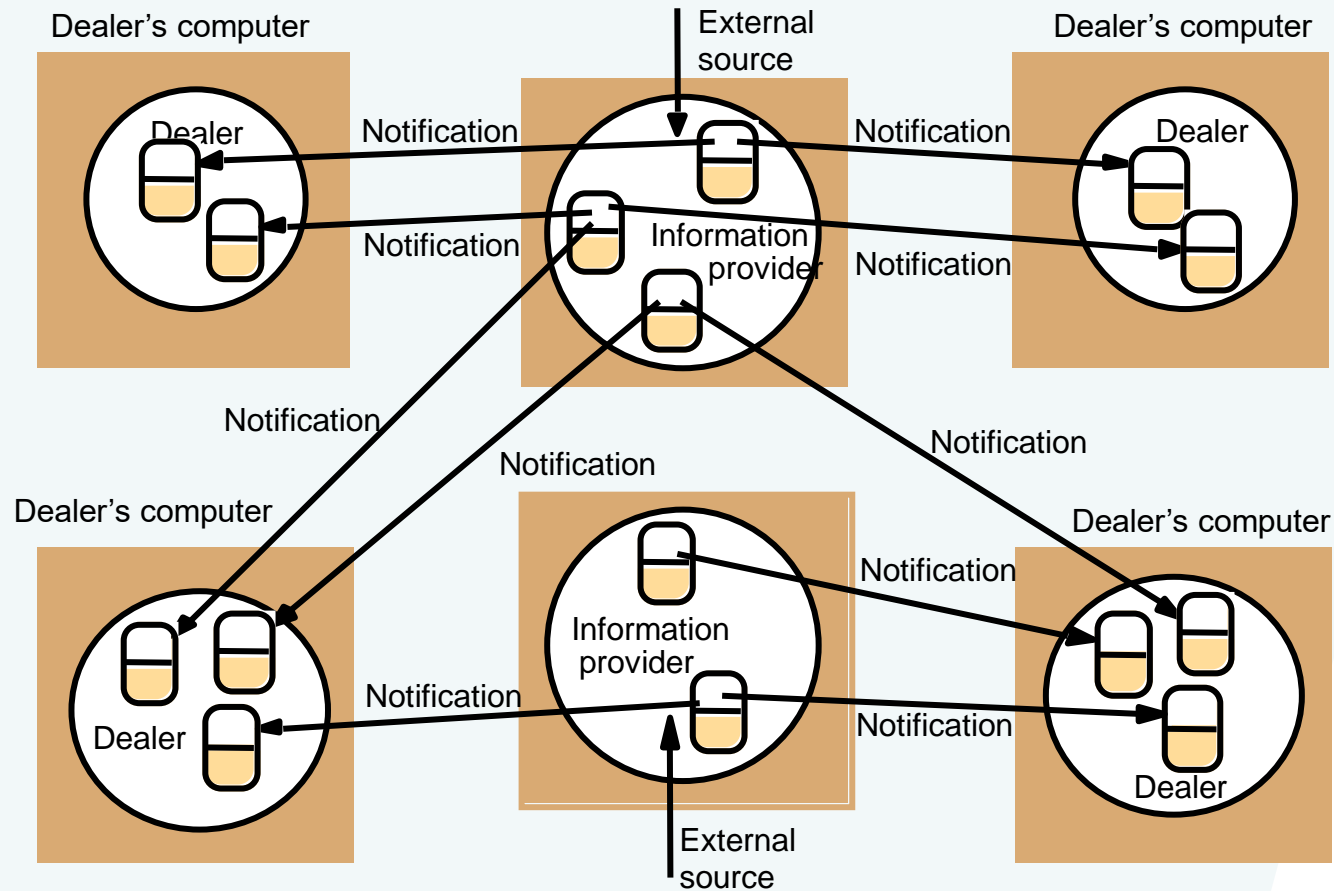
Architecture de JGroups



Java class *FireAlarmJG*

```
import org.jgroups.JChannel;
public class FireAlarmJG {
    public void raise() {
        try {
            JChannel channel = new JChannel();
            channel.connect("AlarmChannel");
            Message msg = new Message(null, "Fire!");
            channel.send(msg);
        }
        catch(Exception e) {
        }
    }
}
```

Sistema de sala de negociación

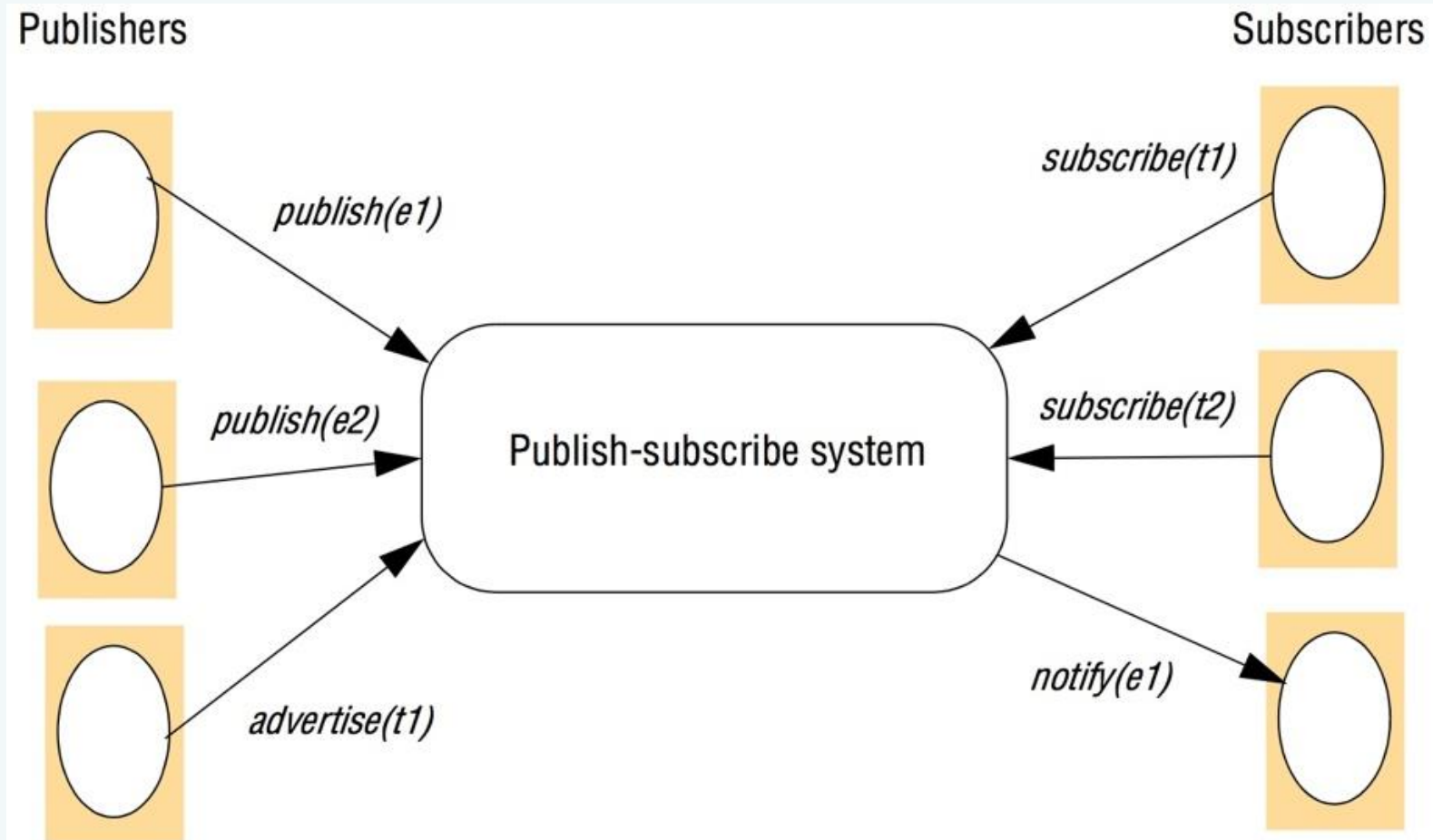




Publicador - Suscriptor

PU

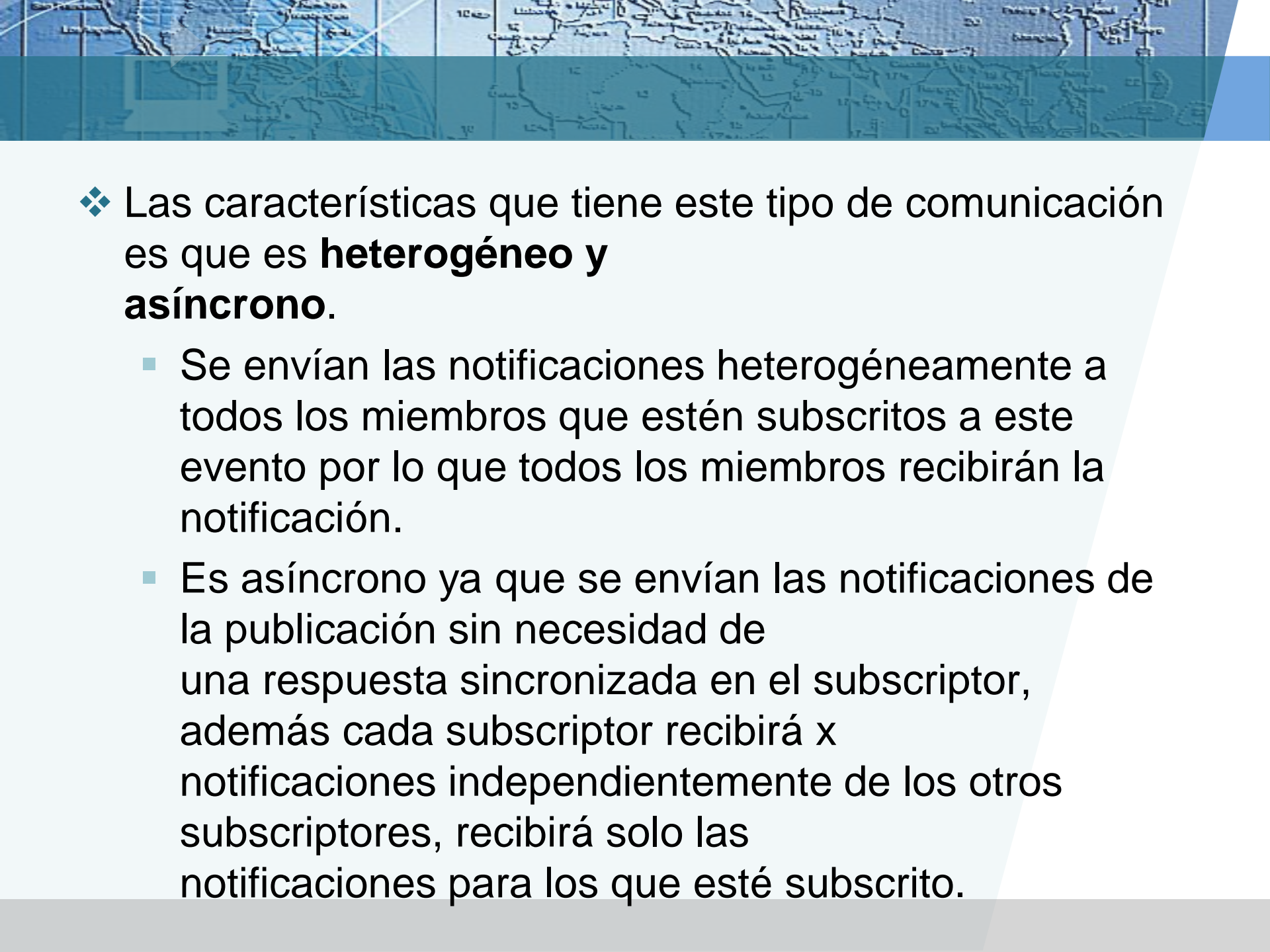
El Paradigma publish-subscribe





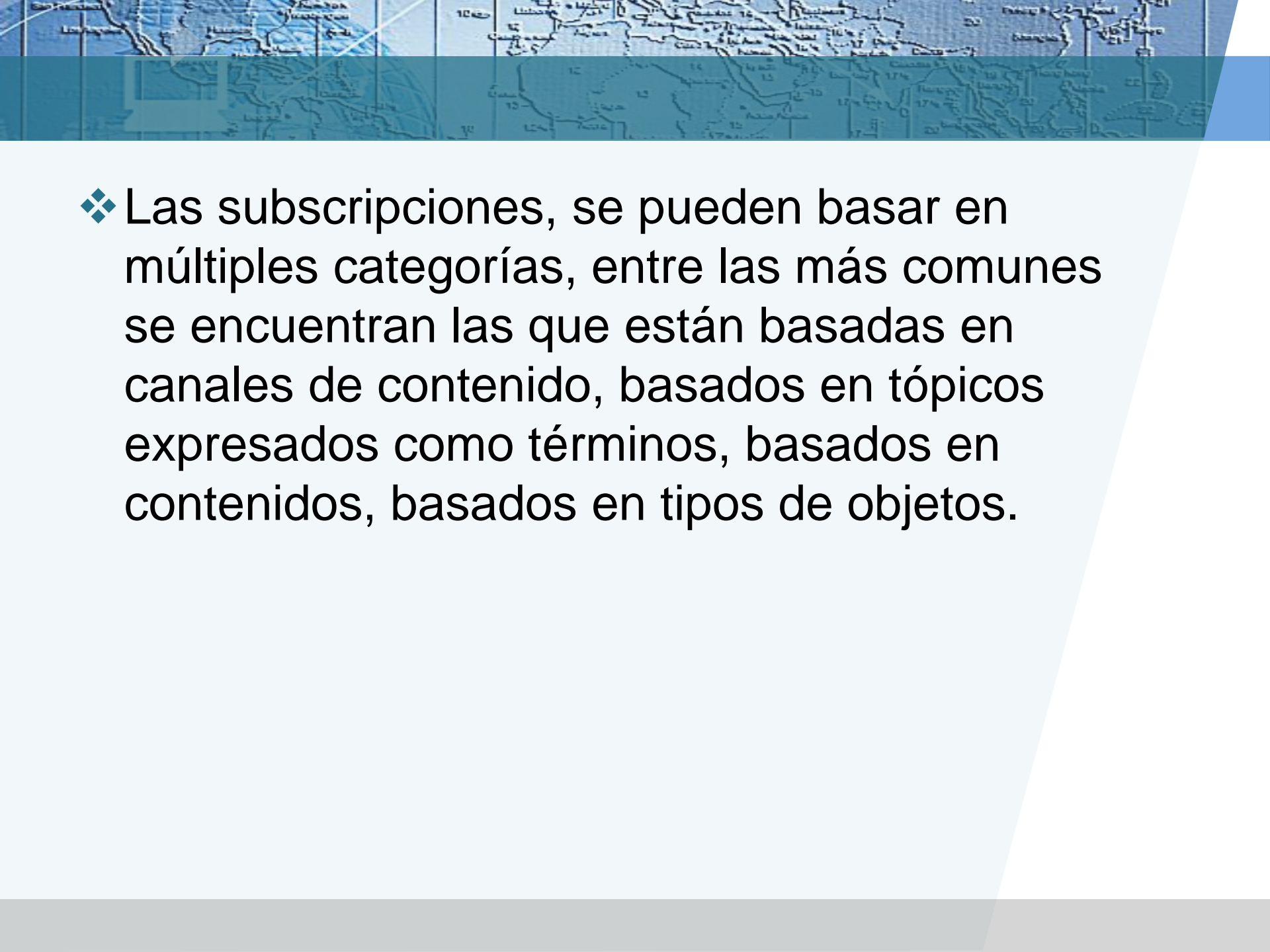
Publicador / Suscriptor


- ❖ Esta **basado en un sistema de publicación de eventos** al que están suscritos determinados equipos.
 - Cuando se realiza la subscripción al publicador estos equipos comunican cuales son los eventos a los que ellos quieren subscribirse en particular.
 - Cuando el **publicador publica un nuevo evento**, este envía una notificación a los equipos que están **suscritos a ese evento**.
 - Así pues los equipos que están suscritos reciben la notificación y actualizan los datos que tienen del evento que se acaba de publicar.

A background image showing a map of the Americas, with North and South America visible. The map is in shades of blue and green, with white lines indicating borders and major cities. The top part of the map is partially obscured by a dark blue horizontal bar.

❖ Las características que tiene este tipo de comunicación es que es **heterogéneo y asíncrono**.

- Se envían las notificaciones heterogéneamente a todos los miembros que estén suscritos a este evento por lo que todos los miembros recibirán la notificación.
- Es asíncrono ya que se envían las notificaciones de la publicación sin necesidad de una respuesta sincronizada en el subscriptor, además cada subscriptor recibirá x notificaciones independientemente de los otros subscriptores, recibirá solo las notificaciones para los que esté suscrito.

- 
- A background image showing a map of the Americas, with North and South America visible. The map is in a light blue and white color scheme, with a darker blue overlay on the right side.
- ❖ Las subscripciones, se pueden basar en múltiples categorías, entre las más comunes se encuentran las que están basadas en canales de contenido, basados en tópicos expresados como términos, basados en contenidos, basados en tipos de objetos.

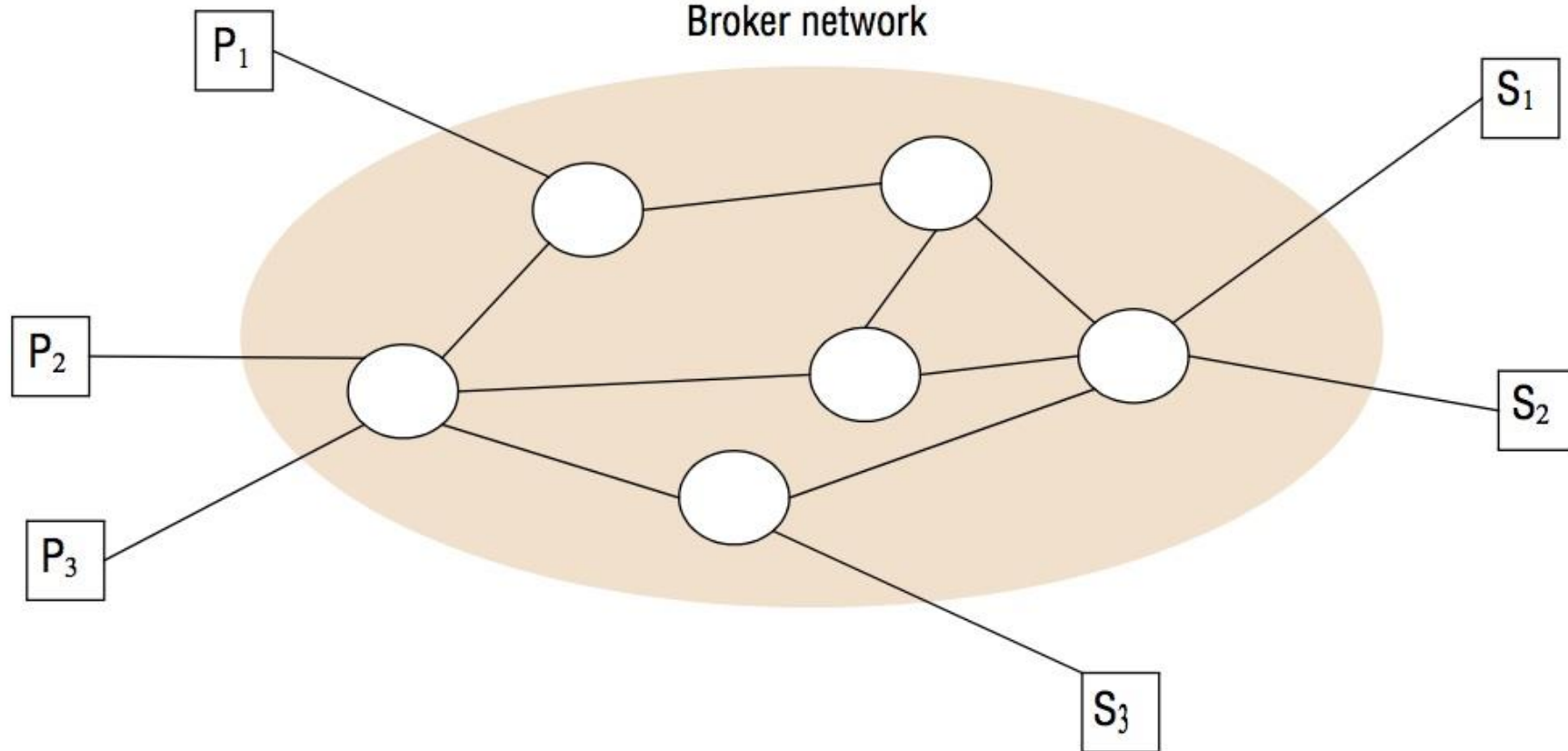
- 
- ❖ Este modelo de comunicación puede utilizarse en **arquitecturas centralizadas y distribuidas**, en el primero habrá un publicador que será el encargado de realizar la notificación al resto de subscriptores, aunque este tipo de arquitectura puede contribuir a formar cuellos de botella y es poco tolerante a fallos, ya que si falla el publicador el sistema fallará.
En el segundo utilizamos sistemas **peer-to-peer** dónde un mismo equipo puede realizar las tareas de publicador y el subscripción, solventará que se formen cuellos de botella y mejorará la tolerancia a fallos.

Una red de corredores

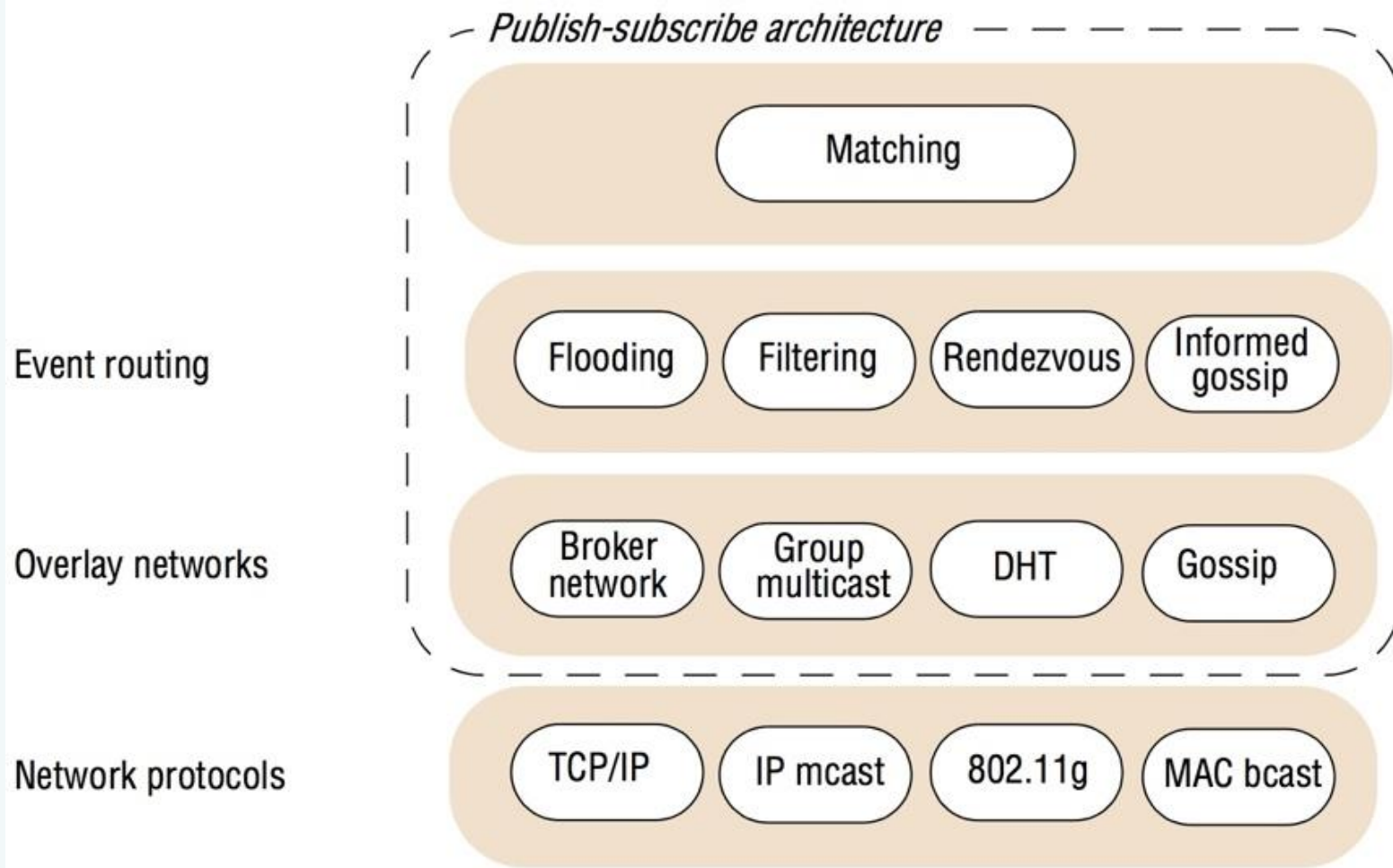
Publishers

Subscribers

Broker network



La arquitectura de los sistemas de publicador-subscriptor



Filtering-based routing

```
upon receive publish(event e) from node x 1  
  matchlist := match(e, subscriptions) 2  
  send notify(e) to matchlist; 3  
  fwddlist := match(e, routing); 4  
  send publish(e) to fwddlist - x; 5  
upon receive subscribe(subscription s) from node x 6  
  if x is client then 7  
    add x to subscriptions; 8  
  else add(x, s) to routing; 9  
  send subscribe(s) to neighbours - x; 10
```

Rendezvous-based routing

```
upon receive publish(event e) from node x at node i  
  rvlist := EN(e);  
  if i in rvlist then begin  
    matchlist := match(e, subscriptions);  
    send notify(e) to matchlist;  
  end  
  send publish(e) to rvlist - i;  
upon receive subscribe(subscription s) from node x at node i  
  rvlist := SN(s);  
  if i in rvlist then  
    add s to subscriptions;  
  else  
    send subscribe(s) to rvlist - i;
```

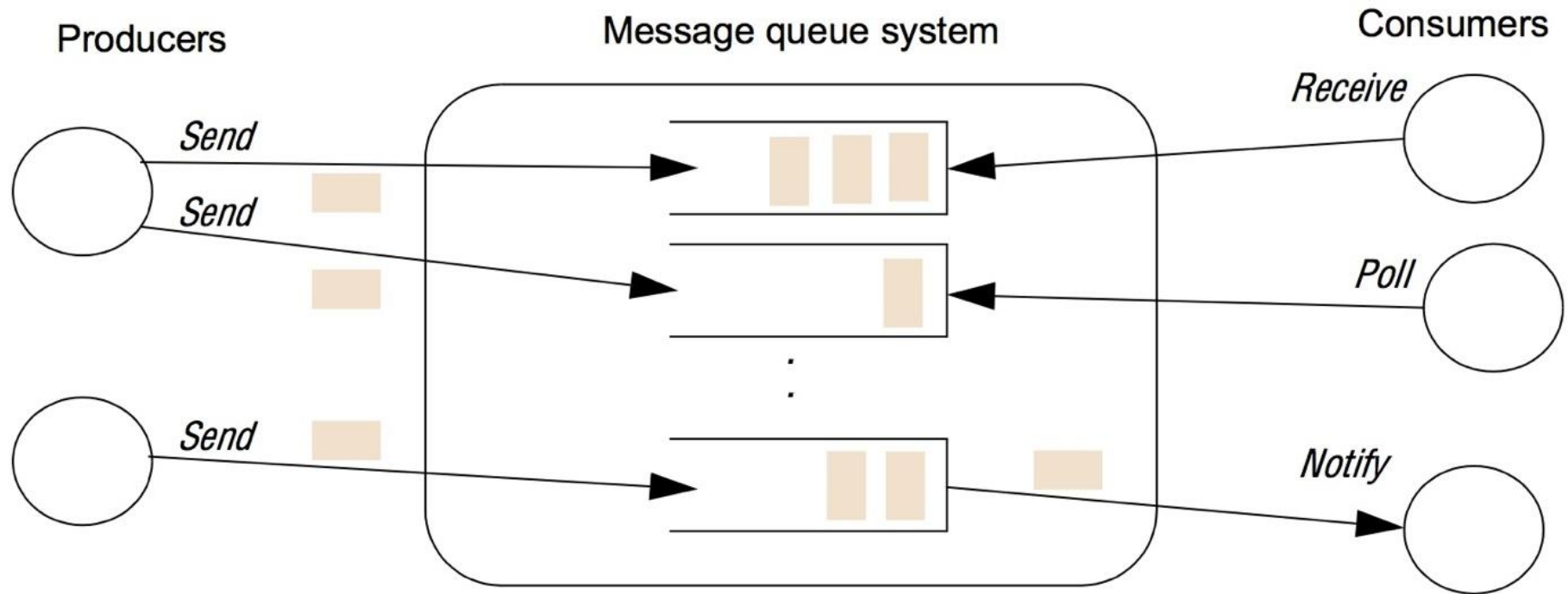

Ejemplo Sistemas publish-subscribe

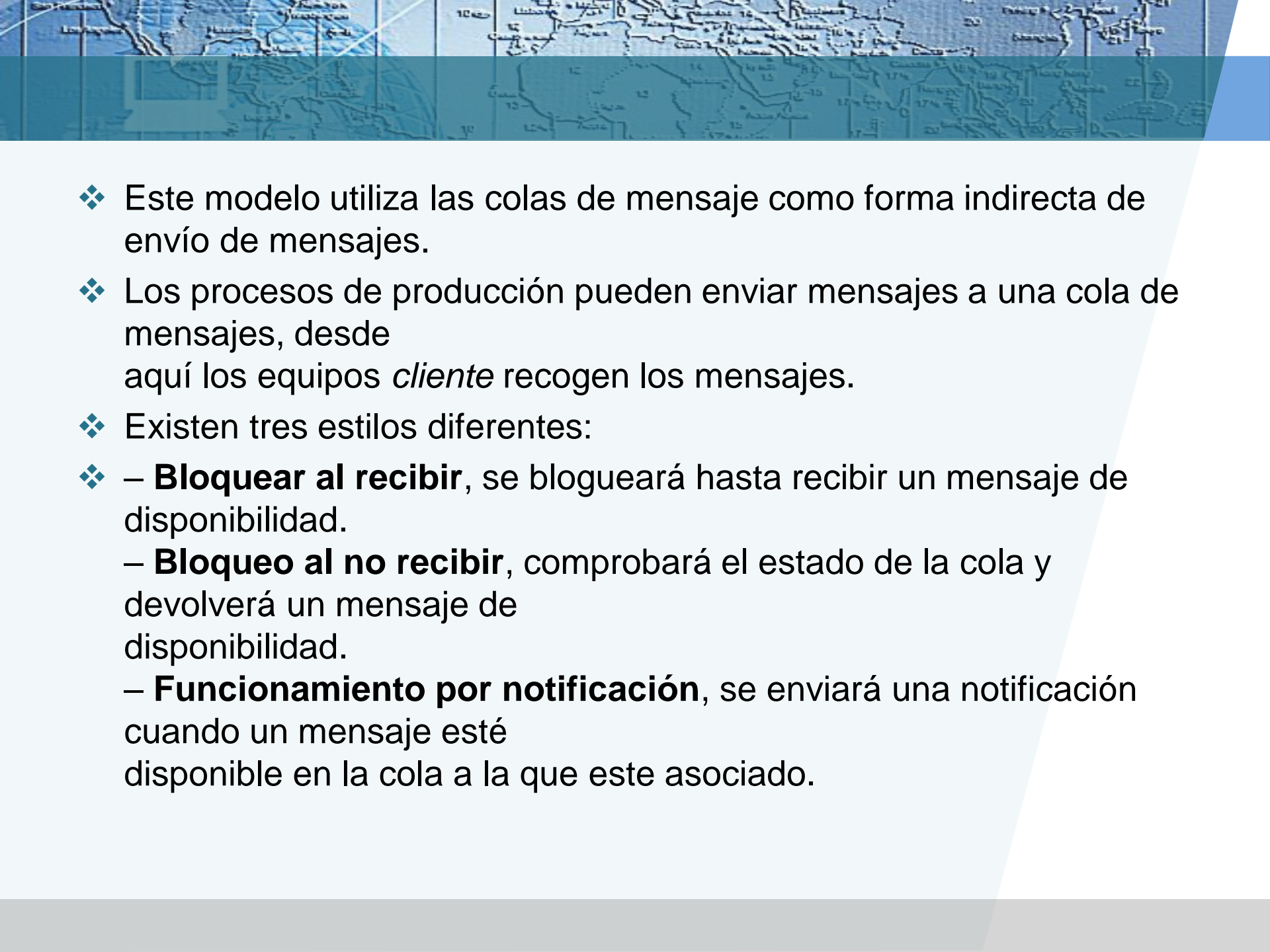
<i>System (and further reading)</i>	<i>Subscription model</i>	<i>Distribution model</i>	<i>Event routing</i>
CORBA Event Service (Chapter 8)	Channel-based	Centralized	-
TIB Rendezvous [Oki <i>et al.</i> 1993]	Topic-based	Distributed	Filtering
Scribe [Castro <i>et al.</i> 2002b]	Topic-based	Peer-to-peer (DHT)	Rendezvous
TERA [Baldoni <i>et al.</i> 2007]	Topic-based	Peer-to-peer	Informed gossip
Siena [Carzaniga <i>et al.</i> 2001]	Content-based	Distributed	Filtering
Gryphon [www.research.ibm.com]	Content-based	Distributed	Filtering
Hermes [Pietzuch and Bacon 2002]	Topic- and content-based	Distributed	Rendezvous and filtering
MEDYM [Cao and Singh 2005]	Content-based	Distributed	Flooding
Meghdoot [Gupta <i>et al.</i> 2004]	Content-based	Peer-to-peer	Rendezvous
Structure-less CBR [Baldoni <i>et al.</i> 2005]	Content-based	Peer-to-peer	Informed gossip




Cola de Mensajes

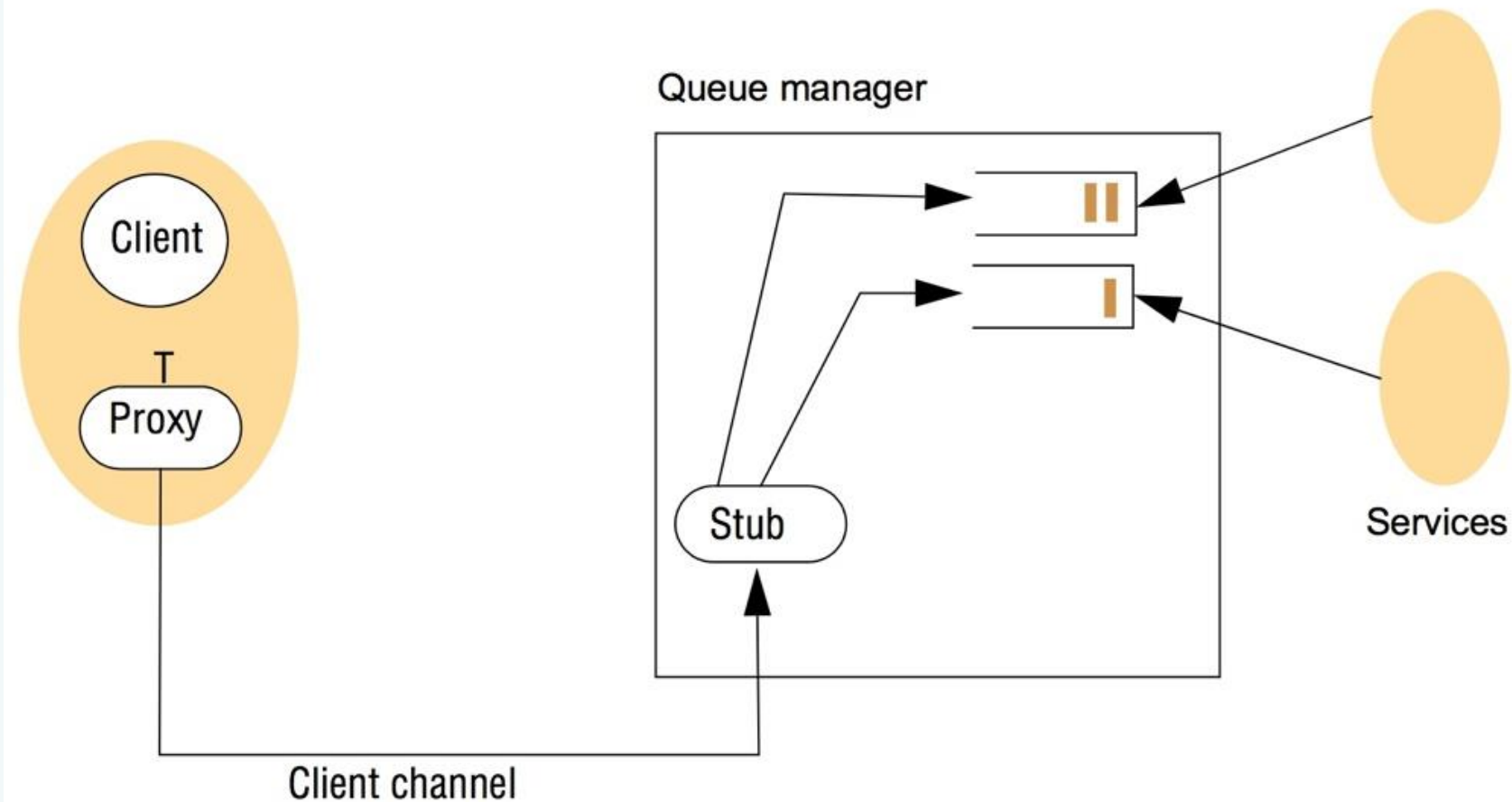
El paradigma de la cola de mensajes.



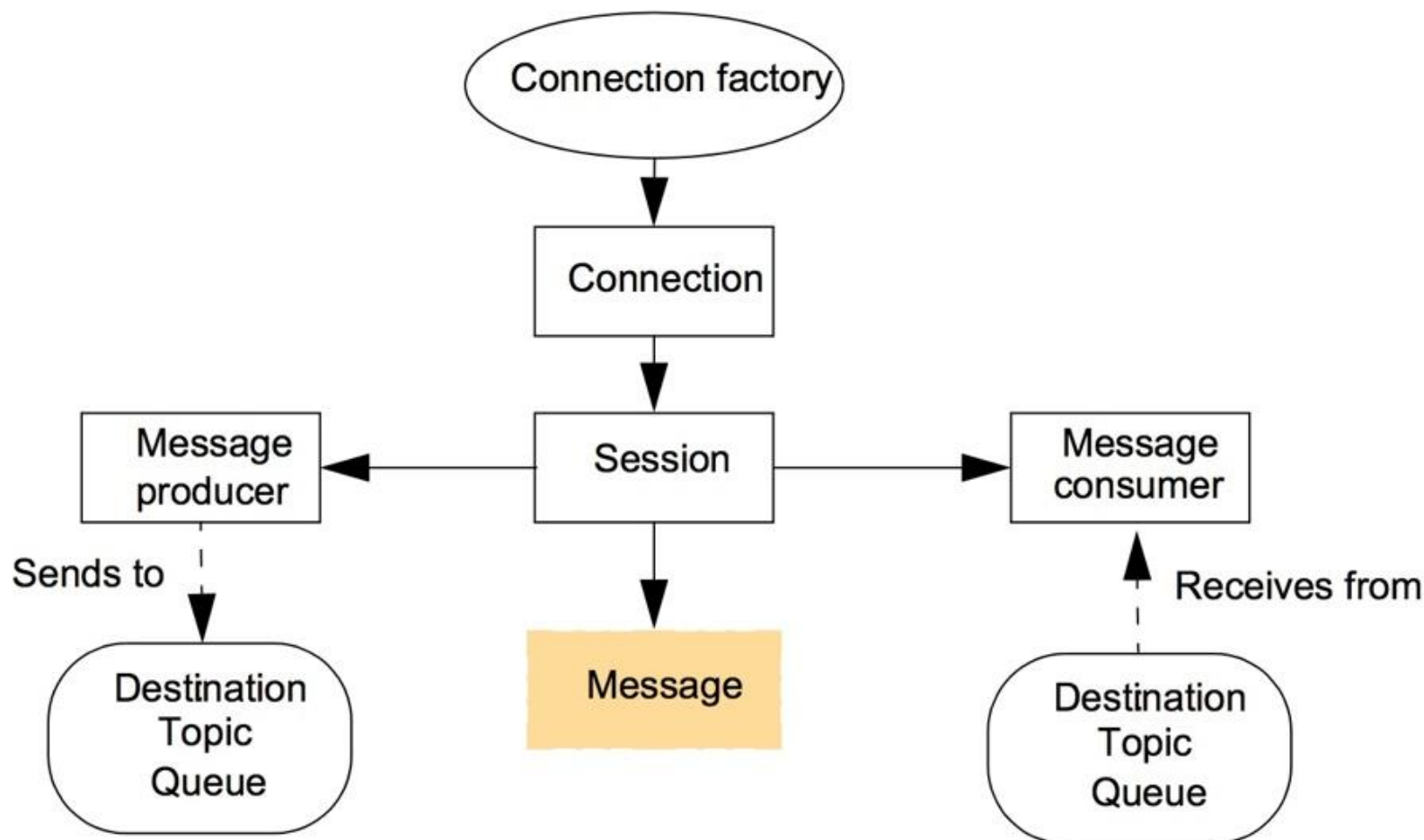
- 
- A faint, blue-toned map of Europe serves as the background for the slide. It shows major countries, rivers, and coastlines. A dark blue horizontal band runs across the top, and a light blue diagonal shape is on the right side.
- ❖ Este modelo utiliza las colas de mensaje como forma indirecta de envío de mensajes.
 - ❖ Los procesos de producción pueden enviar mensajes a una cola de mensajes, desde aquí los equipos *cliente* recogen los mensajes.
 - ❖ Existen tres estilos diferentes:
 - ❖ – **Bloquear al recibir**, se bloqueará hasta recibir un mensaje de disponibilidad.
 - **Bloqueo al no recibir**, comprobará el estado de la cola y devolverá un mensaje de disponibilidad.
 - **Funcionamiento por notificación**, se enviará una notificación cuando un mensaje esté disponible en la cola a la que este asociado.

- 
- ❖ La mayoría de los sistemas que utilizan la cola de mensajes como modelo de comunicación utilizan el orden **FIFO para ordenar la entrada y salida de mensajes**, no obstante se pueden utilizar otros sistemas de ordenación que corresponda mejor con lógica de la aplicación.
 - ❖ Muchos de los sistemas disponibles ofrecen el envío y la recepción de mensajes contenida dentro de una transacción, para favorecer la integridad y poder realizar una transacción completa o descartarla completamente.
 - ❖ También es conveniente que soporten los cambios de codificación de mensaje, para poder transformarlos de codificación e incluso entre diferentes sistemas de intercambios de datos como pueden ser **SOAP**.

Una topología en red simple en WebSphere MQ



El modelo de programación ofertada por JMS



Java class *FireAlarmJMS*

```
import javax.jms.*;
import javax.naming.*;
public class FireAlarmJMS {

    public void raise() {
        try {
            Context ctx = new InitialContext();
            TopicConnectionFactory topicFactory =
                (TopicConnectionFactory)ctx.lookup("TopicConnectionFactory");
            Topic topic = (Topic)ctx.lookup("Alarms");
            TopicConnection topicConn =
                topicConnectionFactory.createTopicConnection();
            TopicSession topicSess = topicConn.createTopicSession(
                false, Session.AUTO_ACKNOWLEDGE);
            TopicPublisher topicPub = topicSess.createPublisher(topic);
            TextMessage msg = topicSess.createTextMessage();
            msg.setText("Fire!");
            topicPub.publish(message);
        } catch (Exception e) {
        }
    }
}
```

Java class *FireAlarmConsumerJMS*

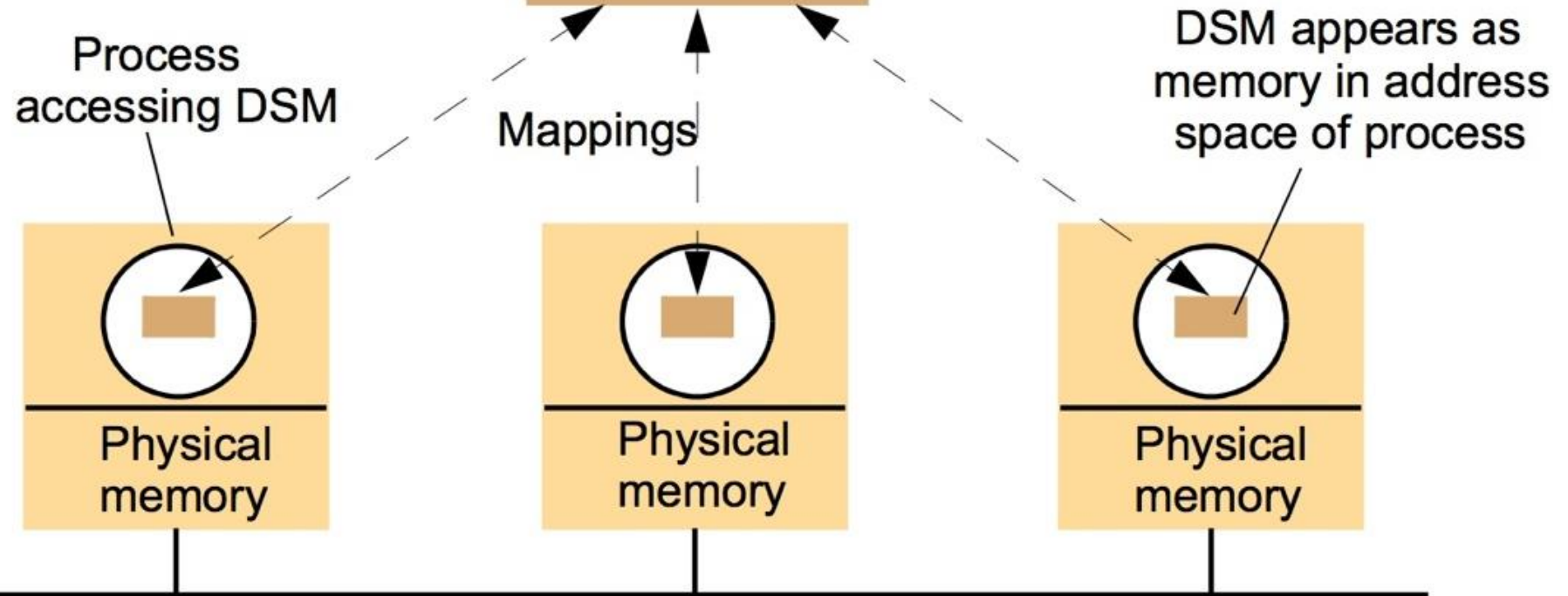
```
import javax.jms.*; import javax.naming.*;
public class FireAlarmConsumerJMS
public String await() {
    try {
        Context ctx = new InitialContext(); 1
        TopicConnectionFactory topicFactory = 2
            (TopicConnectionFactory)ctx.lookup("TopicConnectionFactory"); 4
        Topic topic = (Topic)ctx.lookup("Alarms"); 5
        TopicConnection topicConn = 6
            topicConnectionFactory.createTopicConnection(); 7
        TopicSession topicSess = topicConn.createTopicSession(false, 8
            Session.AUTO_ACKNOWLEDGE); 9
        TopicSubscriber topicSub = topicSess.createSubscriber(topic); 10
        topicSub.start(); 11
        TextMessage msg = (TextMessage) topicSub.receive(); 12
        return msg.getText(); 13
    } catch (Exception e) { 14
        return null; 15
    } 16
}
```

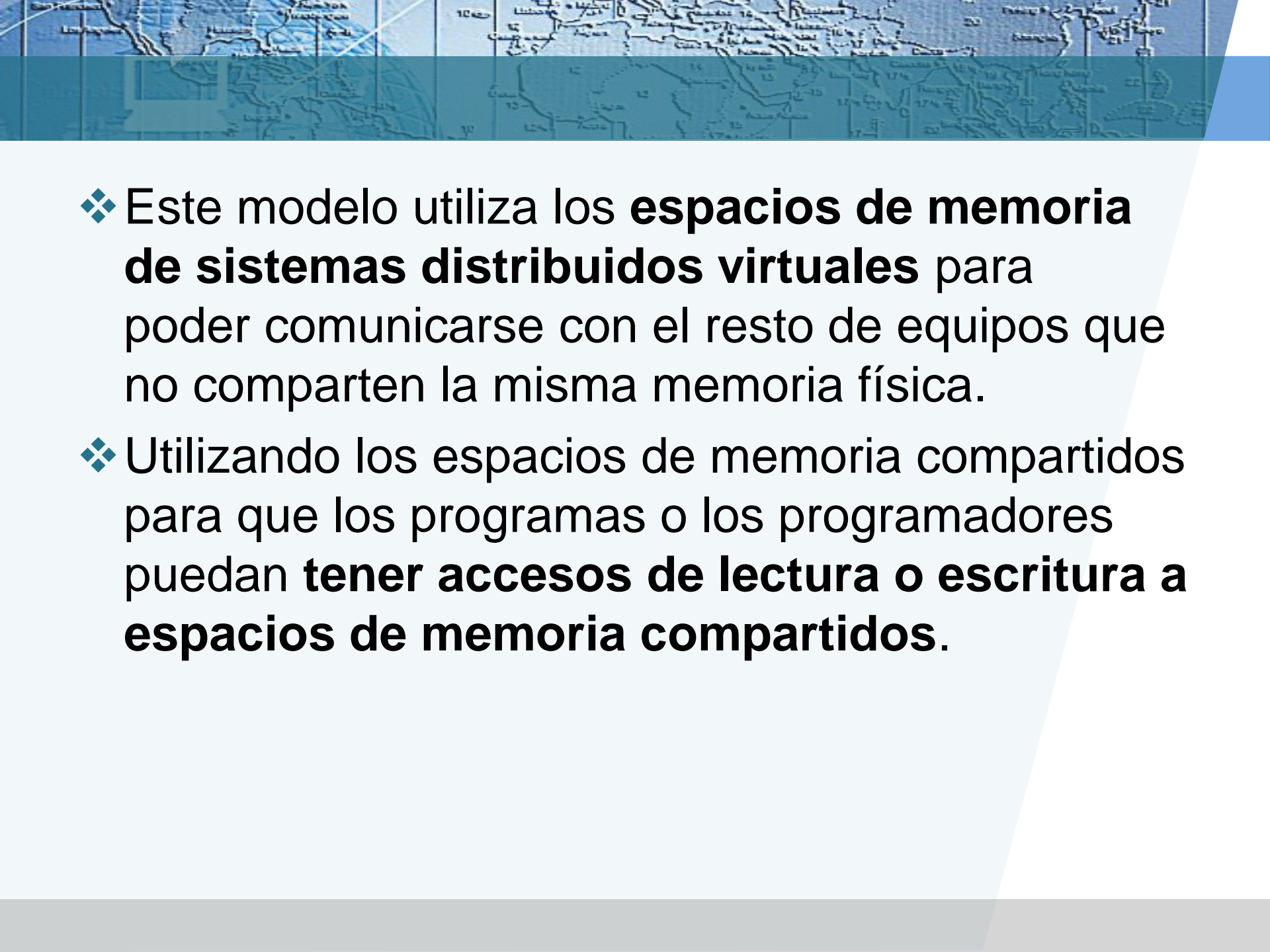



Memoria Compartida

Abstracción memoria compartida distribuida

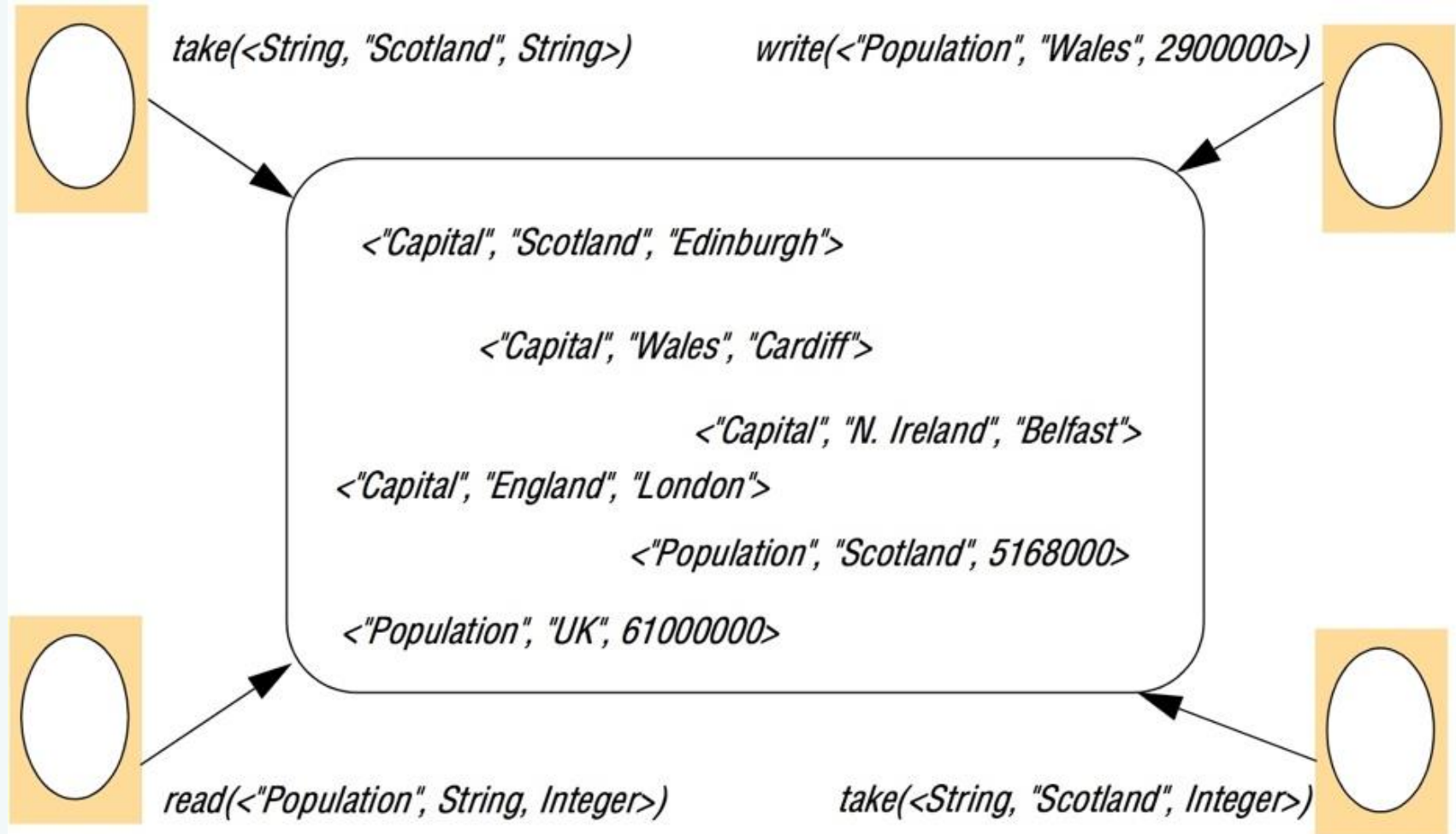
Distributed shared memory



- 
- A background image showing a map of the Americas, with North and South America visible. The map is in a light blue and white color scheme, with a darker blue overlay on the right side.
- ❖ Este modelo utiliza los **espacios de memoria de sistemas distribuidos virtuales** para poder comunicarse con el resto de equipos que no comparten la misma memoria física.
 - ❖ Utilizando los espacios de memoria compartidos para que los programas o los programadores puedan **tener accesos de lectura o escritura a espacios de memoria compartidos**.

- 
- ❖ Los espacios de memoria se dividen en tuplas, en filas en los cuales cada proceso puede leer o escribir dependiendo del caso.
 - ❖ Las propiedades generales que poseen las tuplas, es que tienen **un espacio desacoplado y que el tiempo también esta desacoplado**, por lo tanto una tupla puede tener varios procesos que están enviando datos y varios procesos que están recibiendo esta información. La tupla permanecerá en ese espacio hasta que sea eliminada por lo que tanto el receptor como el emisor no tendrán problemas por solaparse en el tiempo.

Abstracción espacio de tupla



Replication and the tuple space operations [Xu and Liskov 1989]

write

1. The requesting site multicasts the *write* request to all members of the view; 2. On receiving this request, members insert the tuple into their replica and acknowledge this action;
3. Step 1 is repeated until all acknowledgements are received.

read

1. The requesting site multicasts the *read* request to all members of the view;
2. On receiving this request, a member returns a matching tuple to the requestor;
3. The requestor returns the first matching tuple received as the result of the operation (ignoring others); Step 1 is repeated until at least one response is received.

continued on next slide



Replication and the tuple space operations [Xu and Liskov 1989]

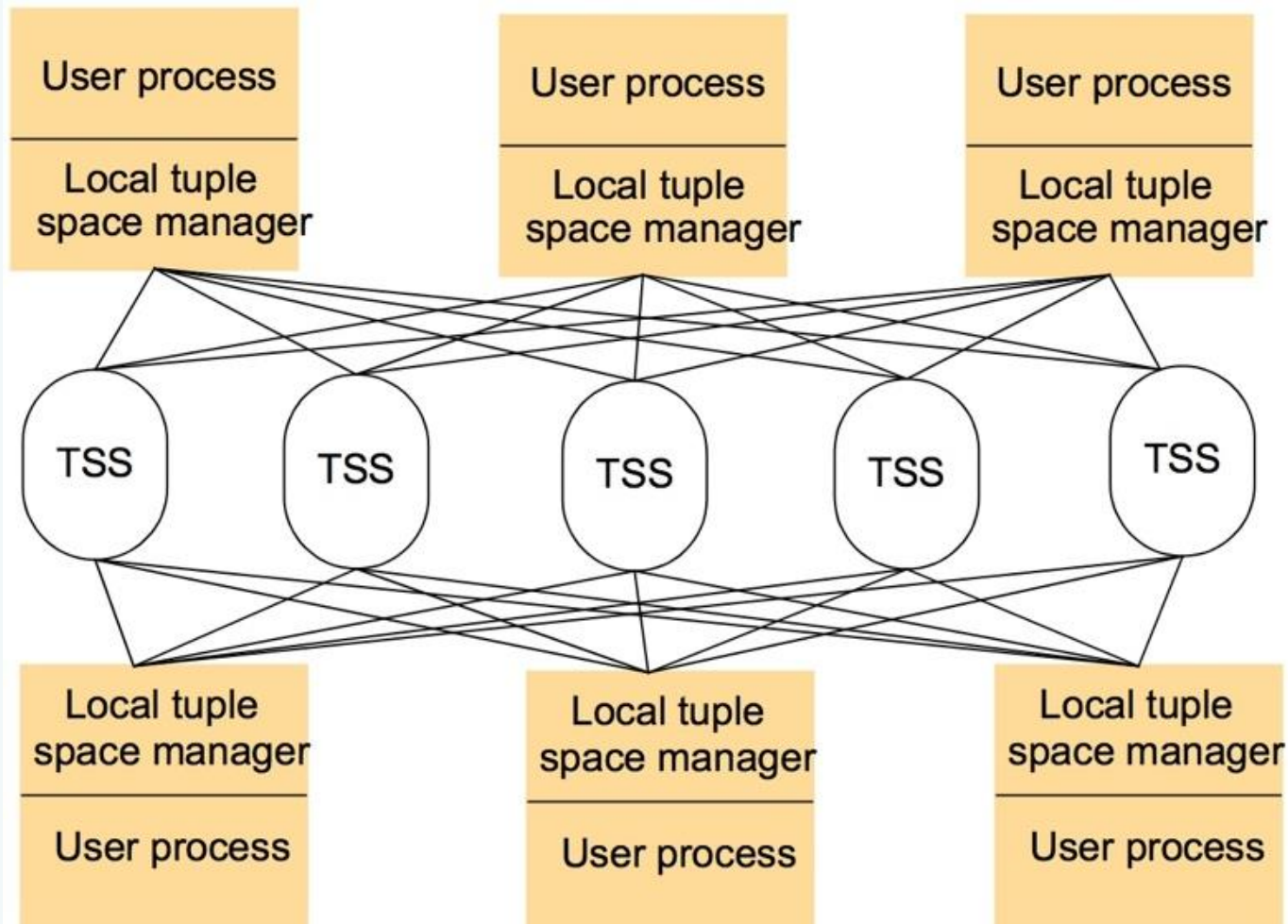
take Phase 1: Selecting the tuple to be removed

1. The requesting site multicasts the *take* request to all members of the view;
2. On receiving this request, each replica acquires a lock on the associated tuple set and, if the lock cannot be acquired, the *take* request is rejected;
3. All accepting members reply with the set of all matching tuples;
4. Step 1 is repeated until all sites have accepted the request and responded with their set of tuples and the intersection is non-null;
5. A particular tuple is selected as the result of the operation (selected randomly from the intersection of all the replies);
6. If only a minority accept the request, this minority are asked to release their locks and phase 1 repeats.

Phase 2: Removing the selected tuple

1. The requesting site multicasts a *remove* request to all members of the view citing the tuple to be removed;
2. On receiving this request, members remove the tuple from their replica, send an acknowledgement and release the lock;
3. Step 1 is repeated until all acknowledgements are received.

Partitioning in the York Linda Kernel



The JavaSpaces API

<i>Operation</i>	<i>Effect</i>
<i>Lease write(Entry e, Transaction txn, long lease)</i>	Places an entry into a particular JavaSpace
<i>Entry read(Entry tmpl, Transaction txn, long timeout)</i>	Returns a copy of an entry matching a specified template
<i>Entry readIfExists(Entry tmpl, Transaction txn, long timeout)</i>	As above, but not blocking
<i>Entry take(Entry tmpl, Transaction txn, long timeout)</i>	Retrieves (and removes) an entry matching a specified template
<i>Entry takeIfExists(Entry tmpl, Transaction txn, long timeout)</i>	As above, but not blocking
<i>EventRegistration notify(Entry tmpl, Transaction txn, RemoteEventListener listen, long lease, MarshalledObject handback)</i>	Notifies a process if a tuple matching a specified template is written to a JavaSpace

Java class *AlarmTupleJS*

```
import net.jini.core.entry.*;
public class AlarmTupleJS implements Entry {
    public String alarmType;
    public AlarmTupleJS() { }
    }
    public AlarmTupleJS(String alarmType) {
        this.alarmType = alarmType;}
    }
}
```

Java class *FireAlarmJS*

```
import net.jini.space.JavaSpace;
public class FireAlarmJS {
    public void raise() {
        try {
            JavaSpace space = SpaceAccessor.findSpace("AlarmSpace");
            AlarmTupleJS tuple = new AlarmTupleJS("Fire!");
            space.write(tuple, null, 60*60*1000);
        } catch (Exception e) {
        }
    }
}
```

Java class *FireAlarmReceiverJS*

```
import net.jini.space.JavaSpace;
public class FireAlarmConsumerJS {
    public String await() {
        try {
            JavaSpace space = SpaceAccessor.findSpace();
            AlarmTupleJS template = new AlarmTupleJS("Fire!");
            AlarmTupleJS recvd = (AlarmTupleJS) space.read(template, null,
                Long.MAX_VALUE);
            return recvd.alarmType;
        }
        catch (Exception e) {
            return null;
        }
    }
}
```


Resumen de sistemas de comunicación indirecta

	<i>Groups</i>	<i>Publish-subscribe systems</i>	<i>Message queues</i>	<i>DSM</i>	<i>Tuple spaces</i>
<i>Space-uncoupled</i>	Yes	Yes	Yes	Yes	Yes
<i>Time-uncoupled</i>	Possible	Possible	Yes	Yes	Yes
<i>Style of service</i>	Communication-based	Communication-based	Communication-based	State-based	State-based
<i>Communication pattern</i>	1-to-many	1-to-many	1-to-1	1-to-many	1-1 or 1-to-many
<i>Main intent</i>	Reliable distributed computing	Information dissemination or EAI; mobile and ubiquitous systems	Information dissemination or EAI; commercial transaction processing	Parallel and distributed computation	Parallel and distributed computation; mobile and ubiquitous systems
<i>Scalability</i>	Limited	Possible	Possible	Limited	Limited
<i>Associative</i>	No	Content-based publish-subscribe only	No	No	Yes