

Módulo 03

La Capa de Transporte

(Pt. 1)



Redes de Computadoras
Depto. de Cs. e Ing. de la Comp.
Universidad Nacional del Sur



Copyright

- Copyright © **2010-2022** A. G. Stankevicius
- Se asegura la libertad para copiar, distribuir y modificar este documento de acuerdo a los términos de la **GNU Free Documentation License**, versión 1.2 o cualquiera posterior publicada por la Free Software Foundation, sin secciones invariantes ni textos de cubierta delantera o trasera
- Una copia de esta licencia está siempre disponible en la página <http://www.gnu.org/copyleft/fdl.html>
- La versión transparente de este documento puede ser obtenida de la siguiente dirección:

<http://cs.uns.edu.ar/~ags/teaching>

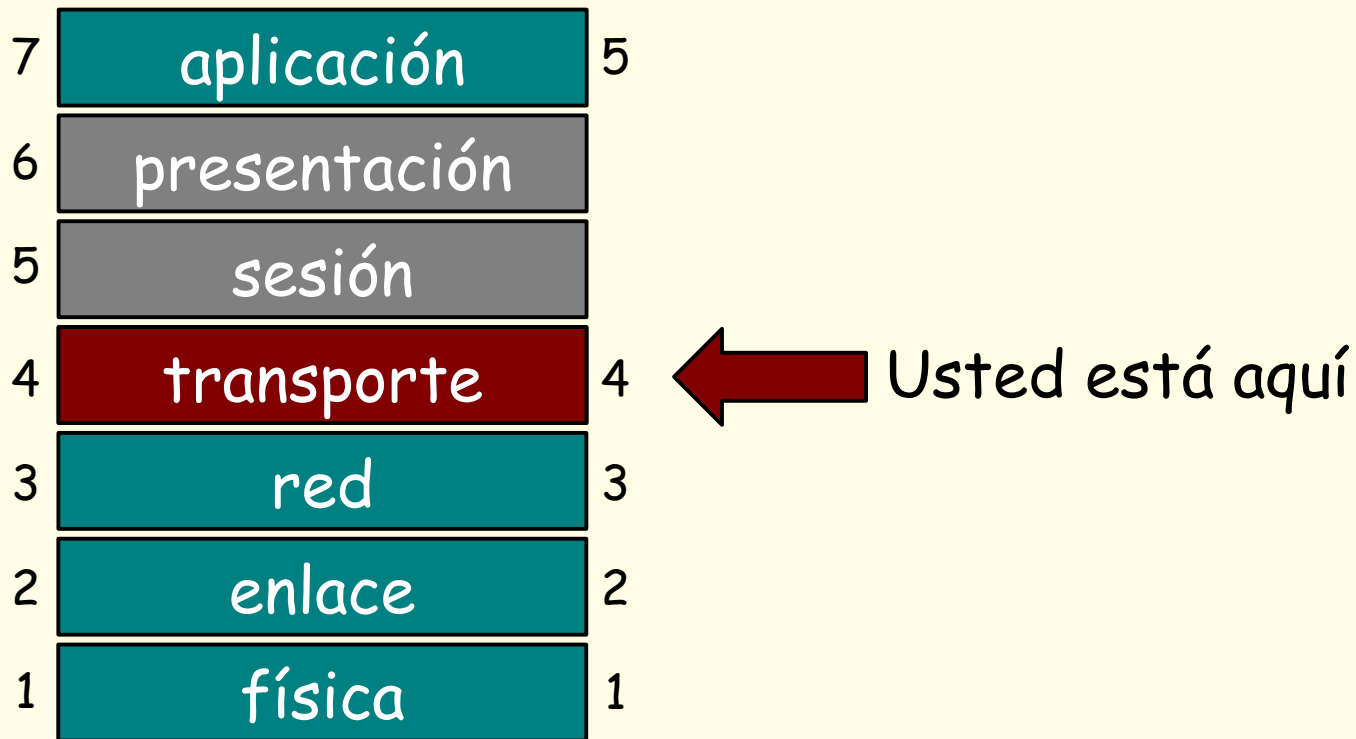


Contenidos

- Servicios y protocolos de la capa de transporte
- Multiplexado y demultiplexado de segmentos
- Transporte no orientado a la conexión (**UDP**)
- Teoría de transporte confiable de datos
- Transporte orientado a la conexión (**TCP**)
- Establecimiento y cierre de conexiones
- Teoría de control de congestión
- Control de congestión en **TCP**

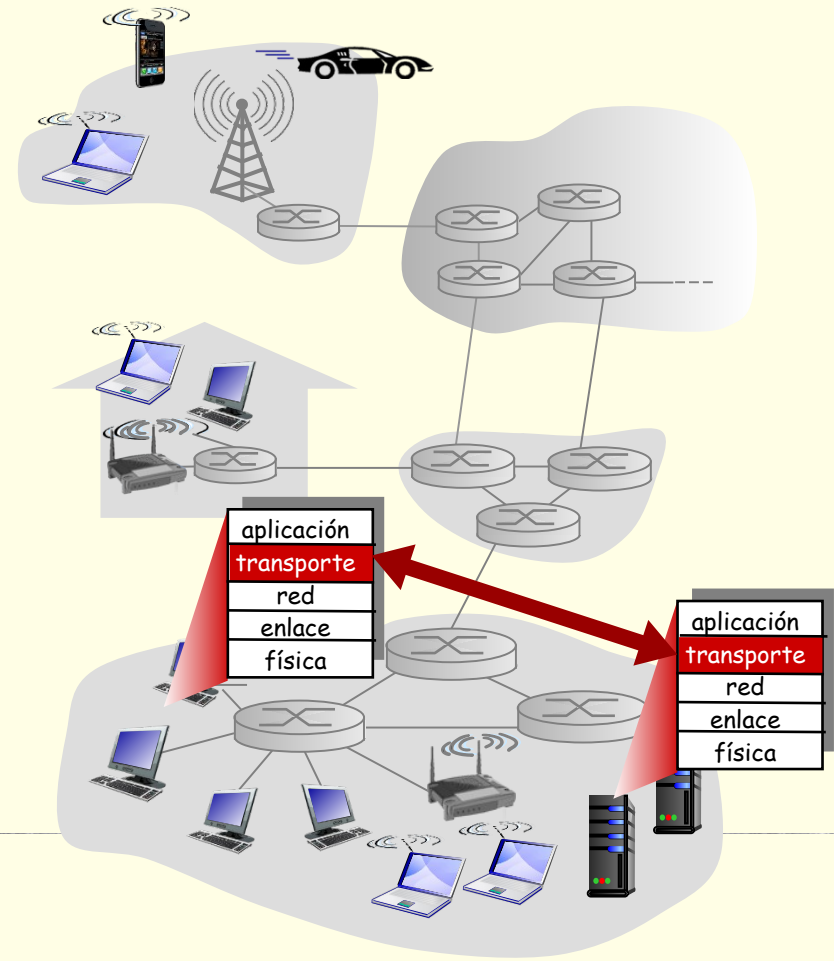


ISO/OSI - TCP/IP



Servicios de transporte

- La capa de transporte provee la **comunicación lógica** entre las diversas aplicaciones de red
- El servicio brindado se implementa a través de los **protocolos de la capa de transporte**
 - ➔ Por caso en internet se cuenta con **TCP** y **UDP**



Protocolos de transporte

- Los protocolos de la capa de transporte corren en las computadoras de la frontera de la red
 - ➔ Cabe enfatizar que los routers en el núcleo de la red usualmente **sólo implementan hasta la capa de red**
- El lado emisor de una comunicación corta los mensajes de las aplicaciones en **segmentos**, los que son pasados a la capa de red
- El lado receptor rearma los mensajes a partir de los segmentos recibidos, que son luego pasados a la capa de aplicaciones



Transporte vs. Red

- La capa de red y la de transporte parecen ser similares, pero en realidad brindan servicios un tanto diferentes:
 - ➔ Por un lado, la capa de red brinda una **conexión lógica punta a punta entre computadoras**
 - ➔ Pero por otro lado, la capa de transporte brinda una **conexión lógica punta a punta entre procesos**
- La capa de transporte no sólo hace uso de los servicios provistos por la capa de red sino que además los **perfecciona y extiende**



Transporte en internet

- Internet cuenta con dos servicios de transporte:
 - **TCP**, orientado a la conexión, que brinda un envío confiable y ordenado de datos e implementa control de flujo y de congestión
 - **UDP**, no orientado a la conexión, que brinda un envío de datos no confiable ni ordenado y tampoco implementa control de flujo ni de congestión
- El servicio de transporte de internet **no tiene forma de asegurar** ni un ancho de banda mínimo ni un retardo máximo



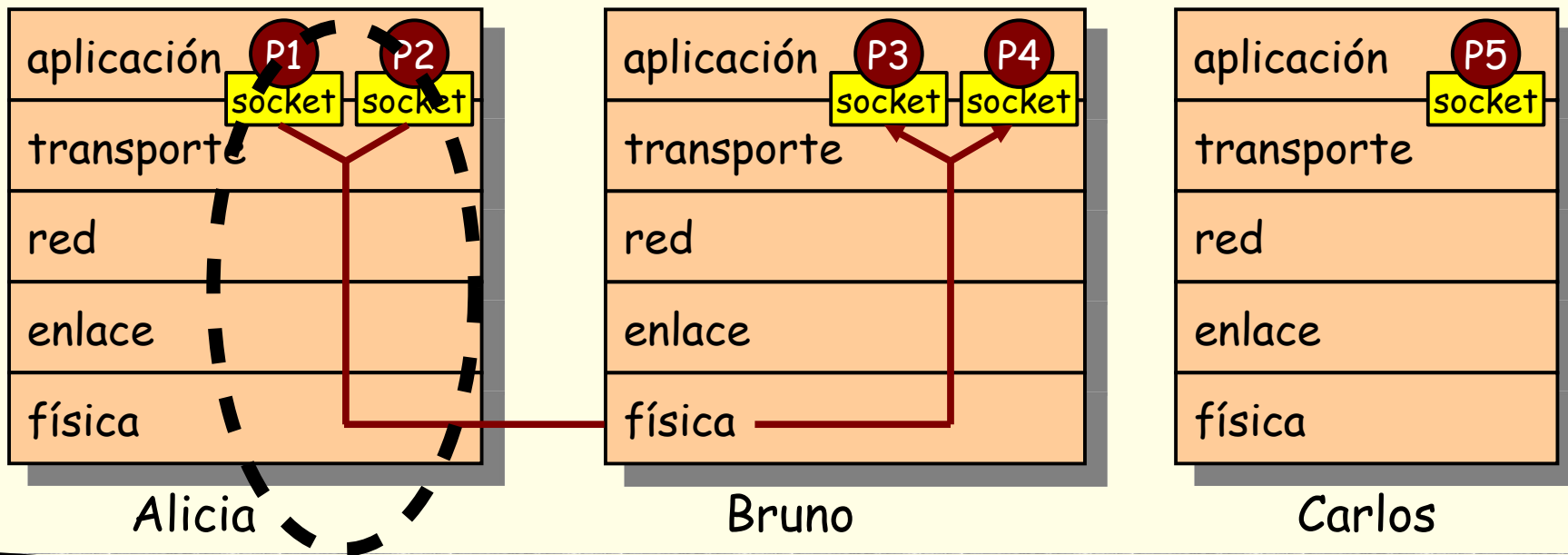
Multiplexado y demultiplexado

- La capa de transporte es la encargada de generalizar el servicio de conexión entre computadoras provisto por la capa de red
- Para la capa de transporte no basta con identificar a una computadora en particular, debe poder identificar a un dado proceso
- El mecanismo que lleva a cabo esta generalización se denomina **multiplexado** del lado del emisor y **demultiplexado** del lado del receptor



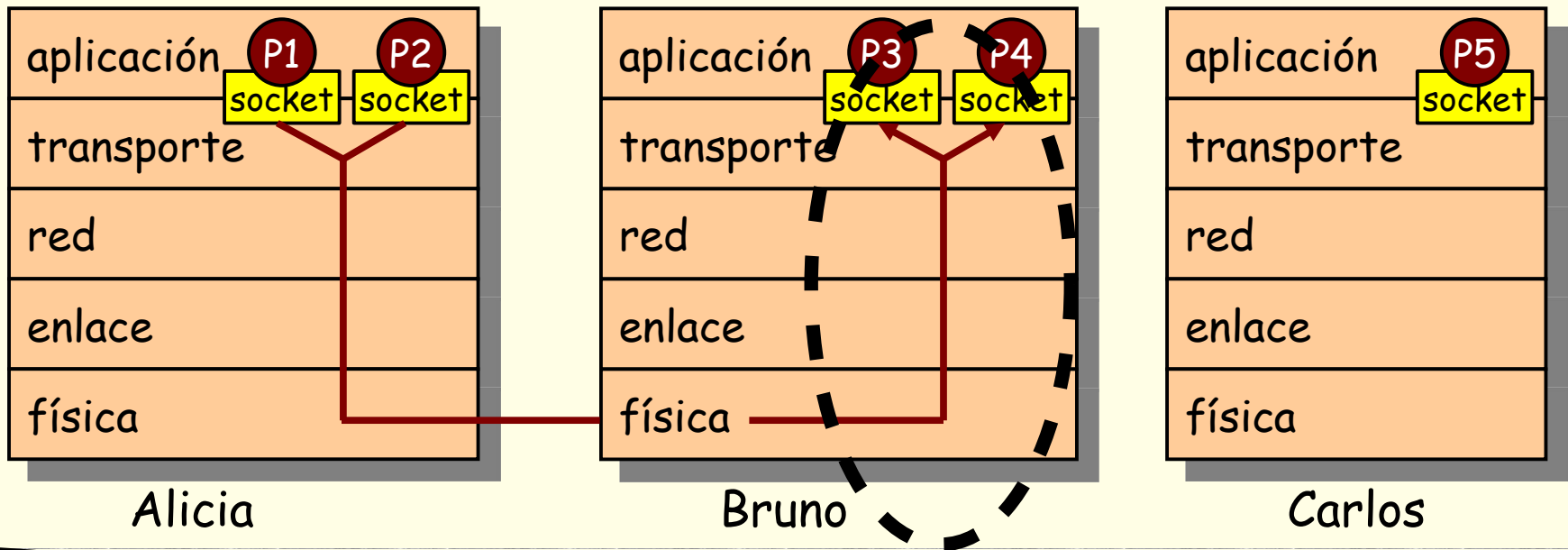
Multiplexado y demultiplexado

- Multiplexado: se realiza en el emisor al juntar mensajes de los sockets con un encabezado adicional, el cual contiene información extra que será usada luego para demultiplexar



Multiplexado y demultiplexado

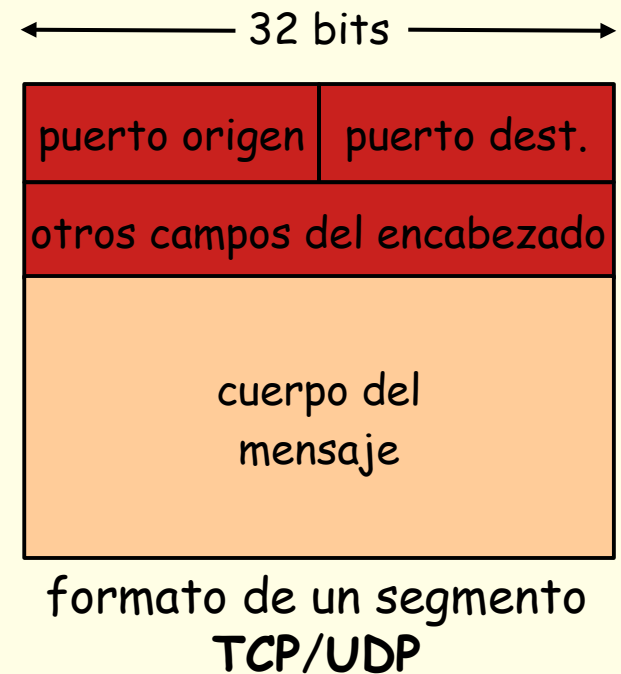
- Demultiplexado: tiene a lugar en el receptor al determinar a qué socket corresponde entregar cada uno de los datos recibidos por un host



Demultiplexado

El proceso de demultiplexado comienza al recibir un **datagrama IP**:

- Cada datagrama tiene un **IP** de origen y de destino
- Contiene exactamente un segmento de la capa de transporte
- Cada segmento especifica un puerto origen y destino
- El **IP** y puerto destino **identifica unívocamente a un socket**



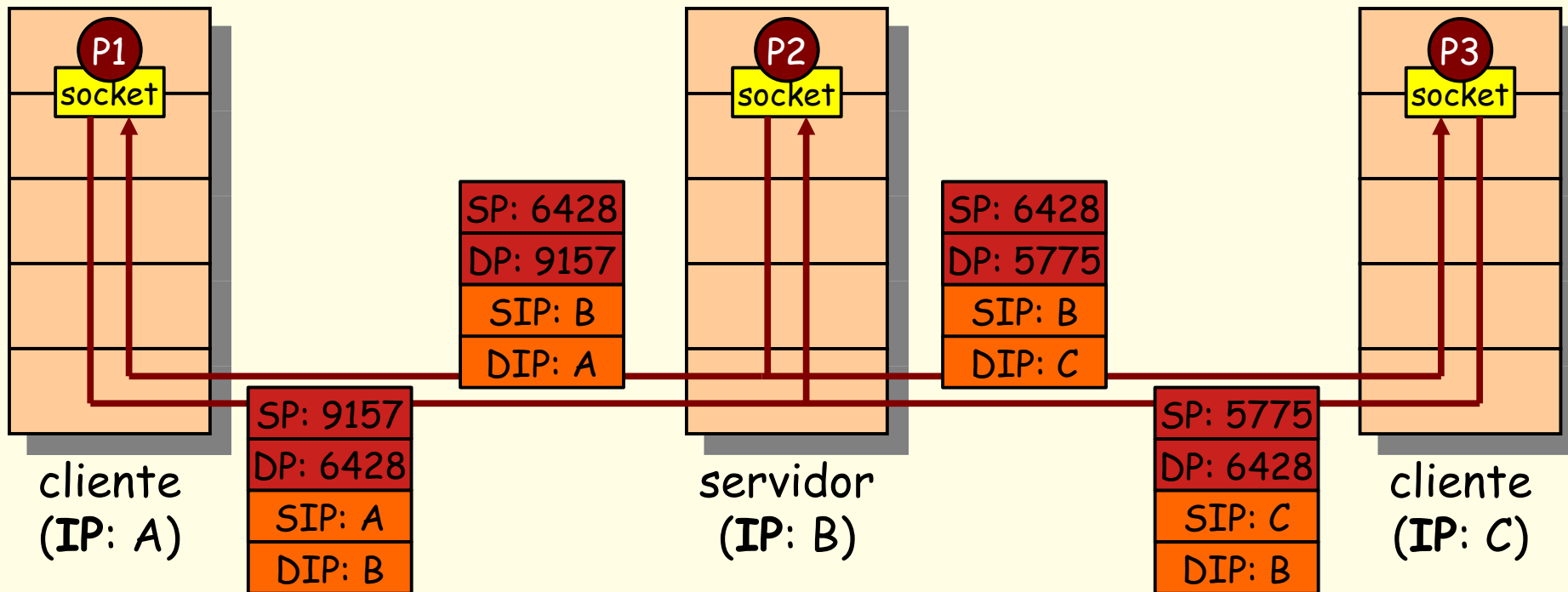
Demultiplexado UDP

- Cada sockets **UDP** se asocia a un número de puerto local determinado
 - ➔ Para identificar un socket **UDP** arbitrario sólo hace falta **una dirección IP** y **un puerto**
- Al recibir un segmento **UDP** se verifica el puerto destino indicado en el segmento y se entrega su contenido al socket asociado a ese puerto
 - ➔ Segmentos originados en máquinas con diversas direcciones **IP** o bien distintos puertos de origen pueden ser entregados al mismo socket **UDP**



Demultiplexado UDP

- Supongamos que el proceso **P2** crea un socket **UDP** en el puerto 6428:



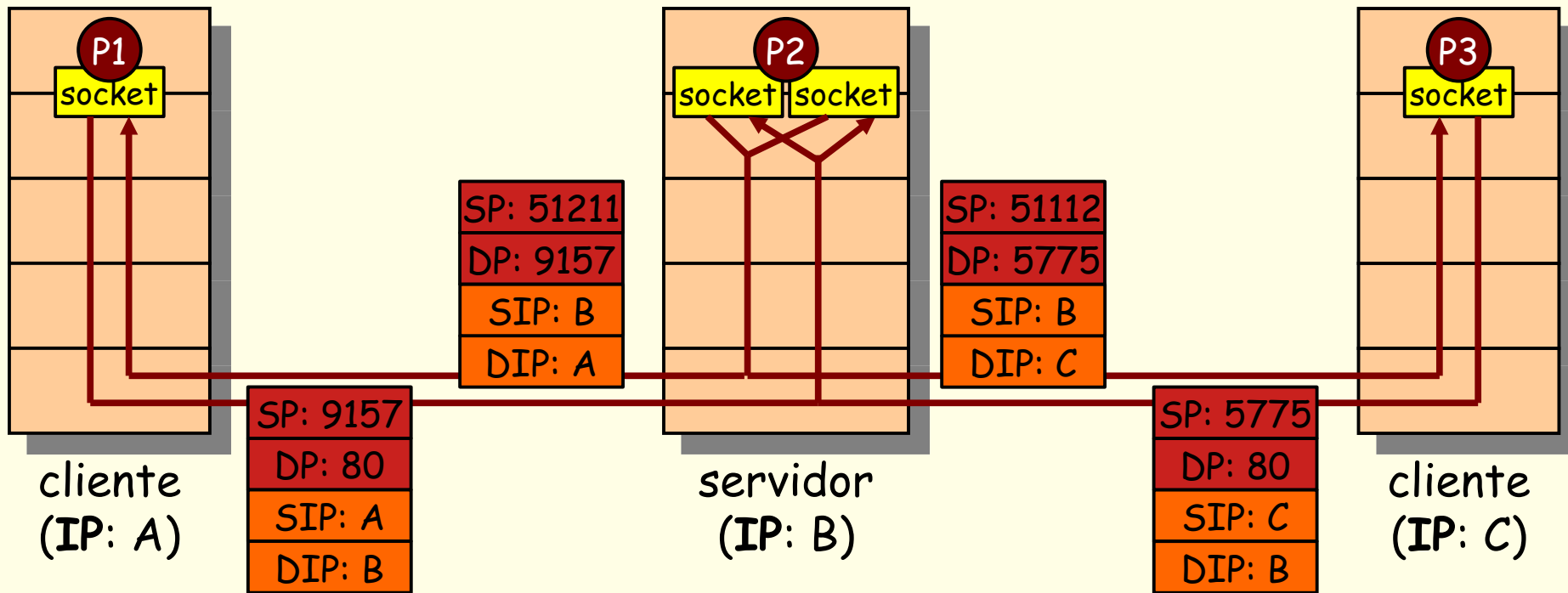
Demultiplexado TCP

- Cada socket **TCP** se asocia a cuatro valores, la dirección **IP** y puerto de origen por un lado y la dirección **IP** y puerto destino por el otro
 - ➔ La computadora que recibe un segmento **TCP** hace uso de esos cuatro valores para determinar cuál de los sockets **TCP** debe recibirlo
- Los servidores naturalmente hacen uso de múltiples sockets, uno para cada uno de los clientes conectados



Demultiplexado TCP

- El proceso **P2** ahora necesita un socket **TCP** independiente para cada cliente:



Protocolo UDP

- El protocolo **UDP** representa el servicio de transporte más elemental que brinda internet
 - Se define formalmente en el **RFC 768**
 - Se basa en el principio “**best effort**”, propio del protocolo **IP** de la capa de red
- Se trata de un protocolo no orientado a la conexión:
 - Cada segmento **UDP** se puede manipular de manera independiente del resto



Protocolo UDP

• ¿Habiendo **TCP**, hace falta **UDP**?

- **UDP** no necesita establecer una conexión antes de comenzar a enviar información
- Es extremadamente simple, no necesita almacenar información acerca del intercambio en curso
- Su encabezado es mucho más sencillo y ocupa menor cantidad de bytes que un encabezado **TCP**
- No implementa control de flujo ni de congestión, el emisor puede enviar la información al ritmo que le plazca, sin limitaciones de ningún tipo



Protocolo UDP

- El protocolo se suele utilizar para hacer streaming de audio y video

- El streaming es tolerante a pérdidas pero requiere retardos acotados

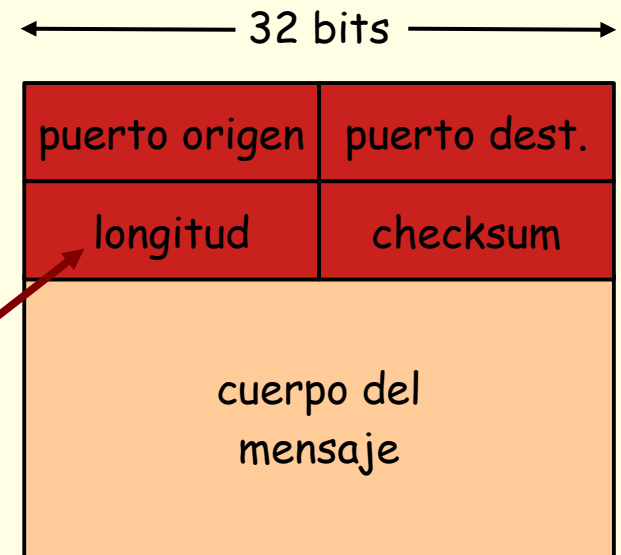
- Otros usos:

- **DNS**

- **SNMP**

- Juegos en línea

largo en bytes del segmento **UDP**, incluyendo los 8 que ocupa el encabezado



formato de un segmento **UDP**



Checksum UDP

- El propósito del campo checksum es **detectar la aparición de bits en error**
- El emisor calcula el checksum correcto:
 - Considera el contenido del segmento como una secuencia de **enteros de 16 bits**
 - El checksum consiste de la **suma en 1-complemento** de esta secuencia de enteros
 - Una vez obtenido el checksum correcto lo escribe en el campo correspondiente del encabezado **UDP**

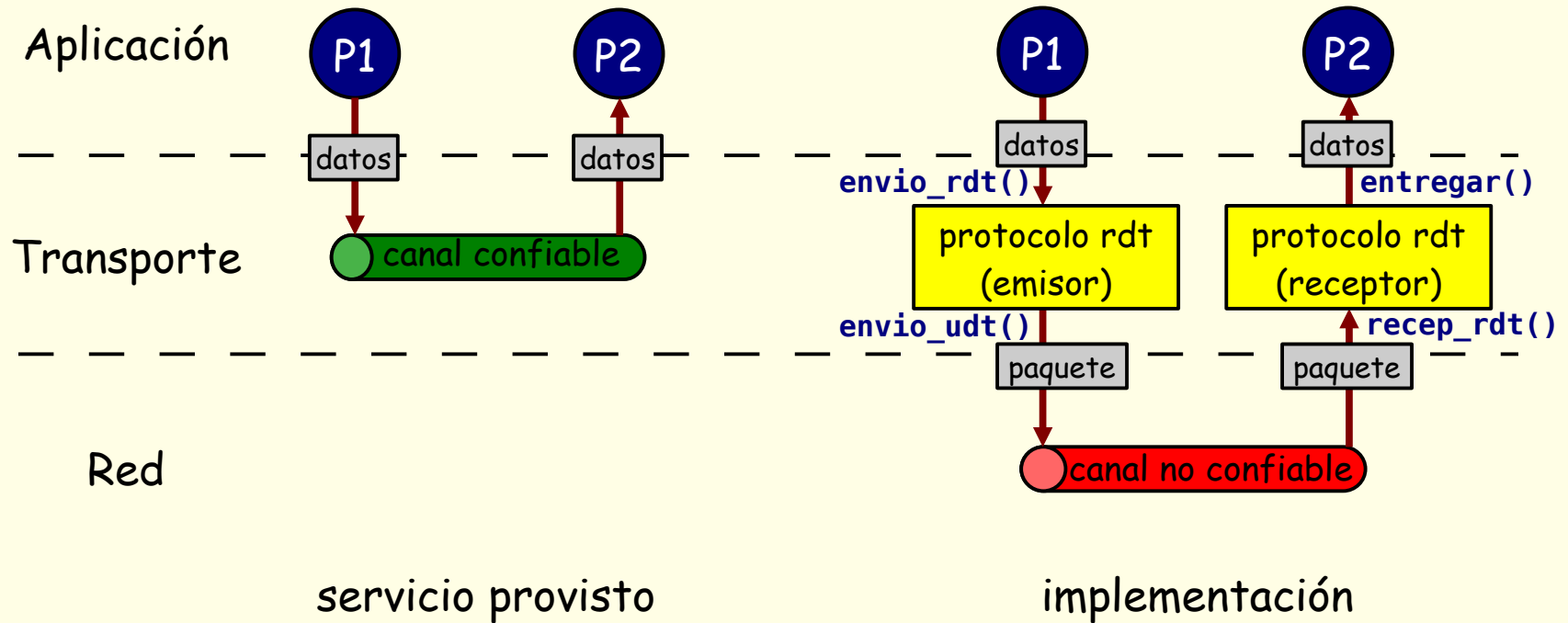


Checksum UDP

- El receptor debe verificar el checksum:
 - Recomputa el checksum del segmento que acaba de recibir
 - Compara el valor obtenido con el checksum registrado en el campo del encabezado
 - Si difieren **se detectó uno o más errores** en la transmisión
 - En contraste, si son iguales **no se han detectados errores en la transmisión** (lo cual no significa que no se hayan producido uno o más errores!)



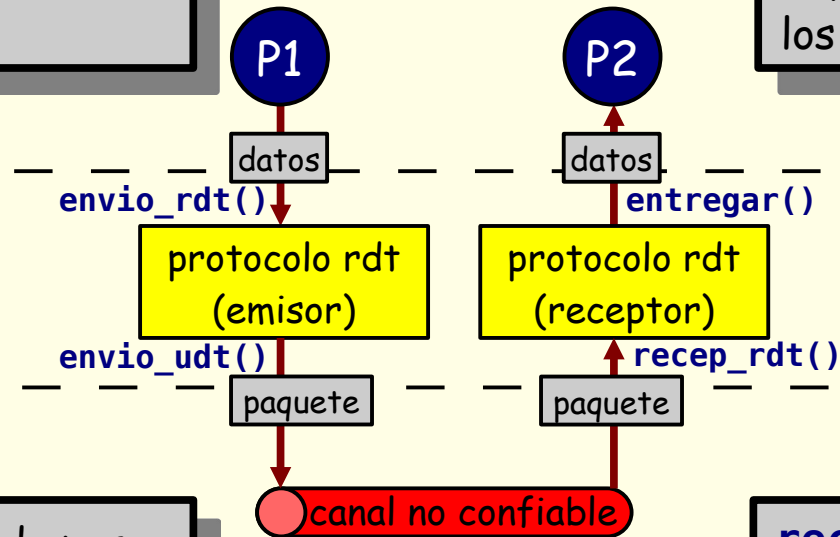
Comunicación confiable



Comunicación confiable

envio_rdt(): invocado desde arriba, suministra los datos a ser enviados

entregar(): invocado por el protocolo para entregar los datos recibidos



envio_udt(): invocado por el protocolo para transferir un paquete a través del canal no confiable

recep_rdt(): invocado por el protocolo cuando llega un nuevo paquete al receptor



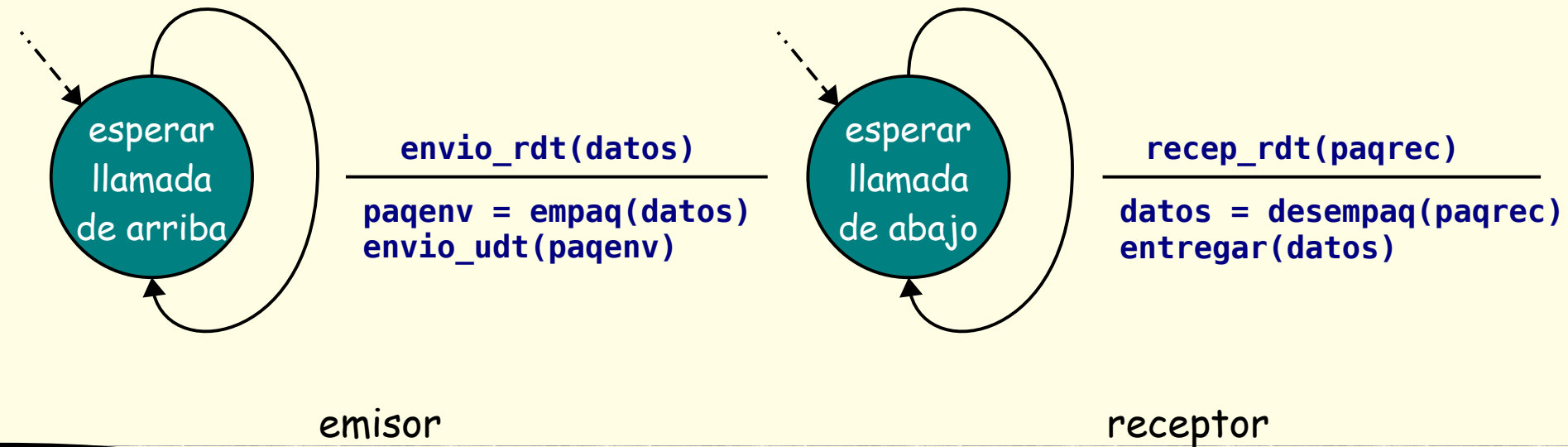
Protocolo RDT

- A continuación analizaremos cómo se debería definir el protocolo **RDT** para poder brindar un **servicio de transferencia confiable de datos**
 - La idea es comenzar con un protocolo básico para luego ir considerando un escenario más realista
 - Emisor y receptor tienen responsabilidades diferentes
 - Sólo consideraremos una **comunicación unidireccional**
 - Las distintas iteraciones del protocolo **RDT** serán definidas a través de **autómatas finitos**



RDT/1.0

- El protocolo **RDT/1.0** permite el envío confiable de datos **a través de un canal confiable**
 - El canal no pierde información ni introduce errores
 - Un autómata para el emisor y otro para el receptor



RDT/2.0

- El protocolo **RDT/2.0** permite envío confiable de datos a través de un canal que puede causar errores a nivel de los bits
 - El canal sigue sin perder información, sólo introduce errores a nivel de bit
 - Los errores pueden ser detectados por el campo checksum del encabezado **UDP**
 - No obstante, con detectar el error no basta, se debe incorporar al protocolo algún mecanismo de recuperación ante la detección de un error
 - ¿Qué hacen los humanos ante este tipo de error?

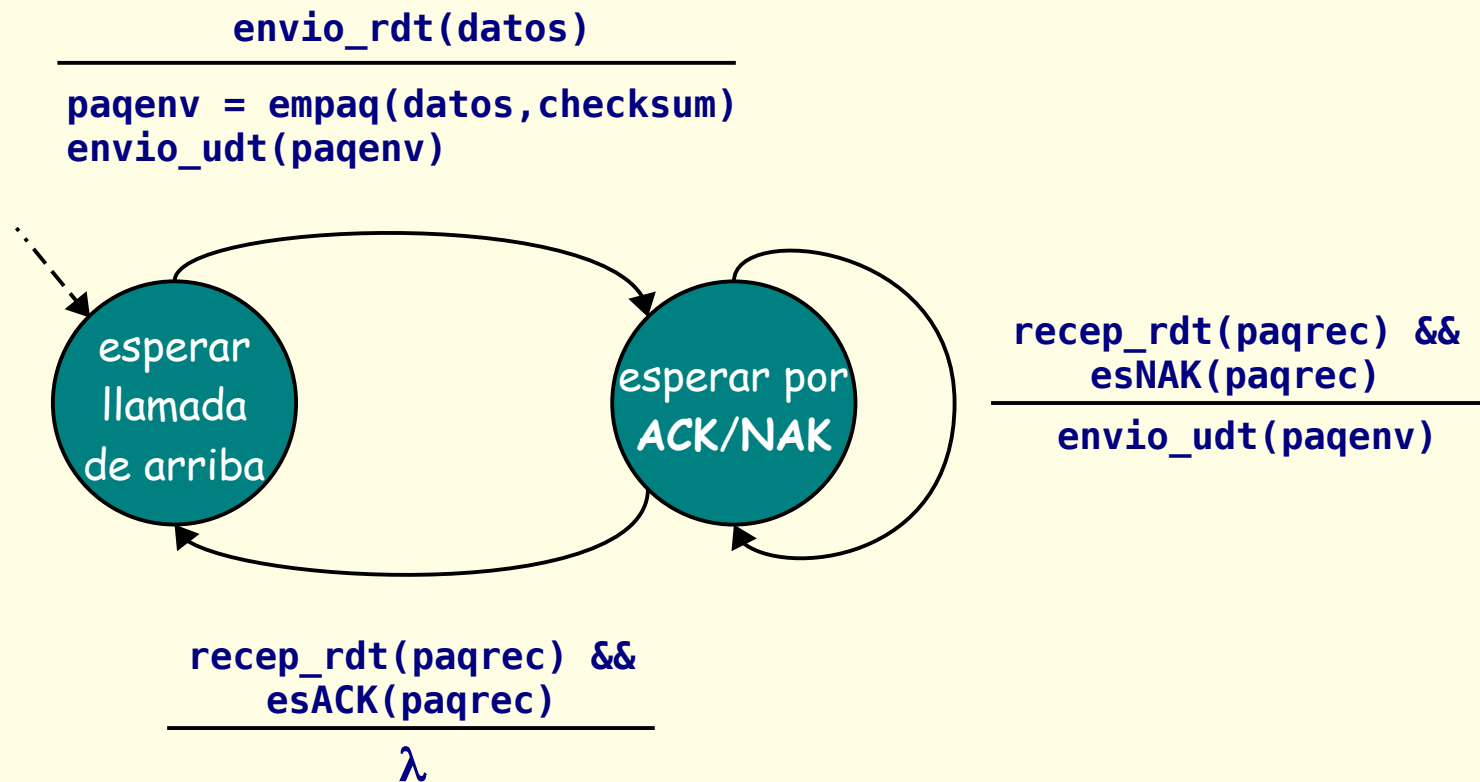


RDT/2.0

- La nueva versión del protocolo hará uso de **confirmaciones de recepción (ACK)** para indicar que el paquete fue recibido sin errores
- Si el paquete llega con errores, se **solicita la retransmisión** del mismo haciendo uso de un mensaje específico (**NAK**)
- El emisor reenvía el último paquete enviado al recibir un **NAK** en vez de un **ACK**
 - ➔ ¿Existe alguna situación involucrando humanos en la que se haga uso de mensajes de **ACK** y/o **NAK**?



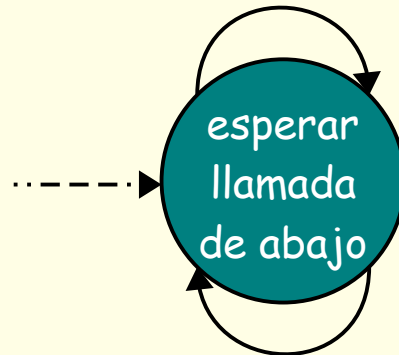
Emisor RDT/2.0



Receptor RDT/2.0

```
recep_rdt(paqrec) &&  
corrupto(paqrec)
```

```
paqenv = crearpaqNAK()  
envio_udt(paqenv)
```



```
recep_rdt(paqrec) &&  
!corrupto(paqrec)
```

```
paqenv = crearpaqACK()  
envio_udt(paqenv)  
datos = desempaq(paqrec)  
entregar(datos)
```



RDT/2.0 envío sin errores

envio_rdt(datos)

paqenv = empaq(datos,checksum)
envio_udt(paqenv)

recep_rdt(paqrec) &&
esNAK(paqrec)

envio_udt(paqenv)

recep_rdt(paqrec) &&
corrupto(paqrec)

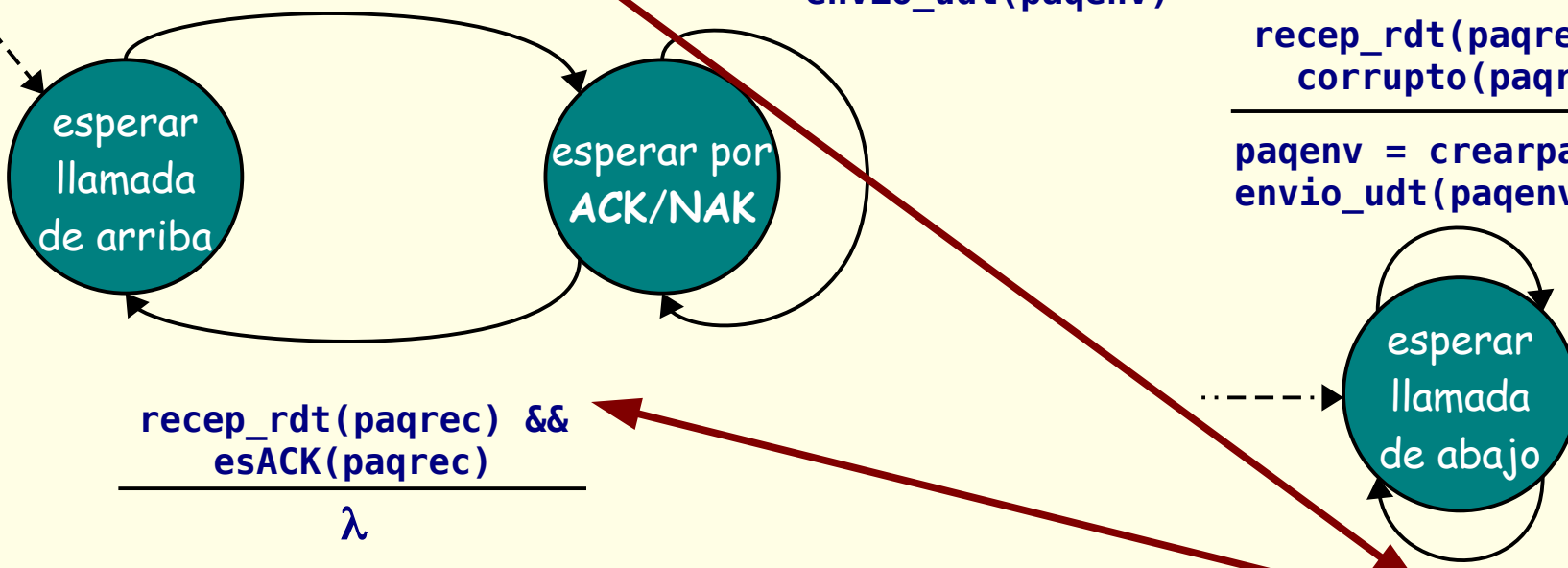
paqenv = crearpaqNAK()
envio_udt(paqenv)

recep_rdt(paqrec) &&
esACK(paqrec)

λ

recep_rdt(paqrec) &&
!corrupto(paqrec)

paqenv = crearpaqACK()
envio_udt(paqenv)
datos = desempaq(paqrec)
entregar(datos)



RDT/2.0 envío con errores

envio_rdt(datos)

paqenv = empaq(datos,checksum)
envio_udt(paqenv)

recep_rdt(paqrec) &&
esNAK(paqrec)

envio_udt(paqenv)

recep_rdt(paqrec) &&
corrupto(paqrec)

paqenv = crearpaqNAK()
envio_udt(paqenv)

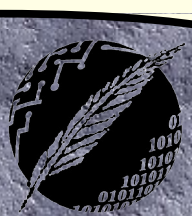
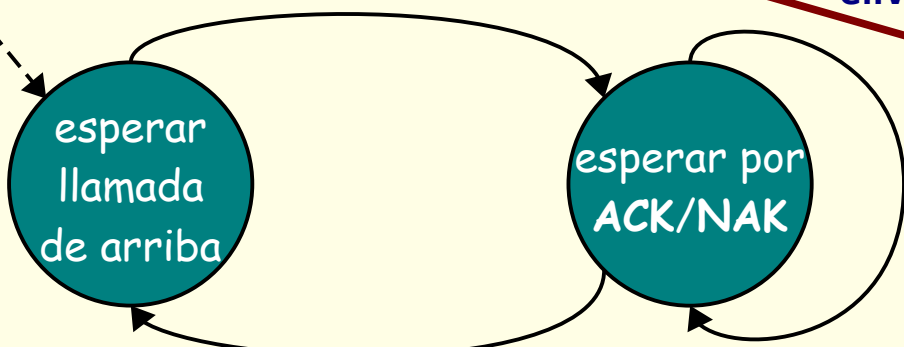
esperar llamada de abajo

recep_rdt(paqrec) &&
esACK(paqrec)

λ

recep_rdt(paqrec) &&
!corrupto(paqrec)

paqenv = crearpaqACK()
envio_udt(paqenv)
datos = desempaq(paqrec)
entregar(datos)



Análisis de RDT/2.0

- **RDT/2.0** es un protocolo tipo “**stop and wait**”
 - El emisor envía un paquete y queda a la espera de que le confirmen su correcta recepción
- No obstante, **RDT/2.0** tiene un defecto fatal:
 - ¿Qué sucede si el canal corrompe el paquete enviado por el receptor conteniendo un **ACK** o un **NAK**?
 - El problema es que el receptor desconoce qué está pasando con el emisor
 - Asumir que se trataba de un **NAK** no es suficiente, puede causar que el receptor acepte un duplicado



Posibles soluciones

- Una posibilidad consiste en **enviar un ACK o un NAK adicional** para que el receptor sepa si el emisor recibió correctamente el mensaje
 - ➔ ¿Qué pasa si se corrompe el **ACK/NAK** del **ACK/NAK**?
- Otra posibilidad es **agregar suficientes bits de código** como para poder corregir estos errores
 - ➔ Podría funcionar, pero no se puede aplicar a canales que pierdan paquetes
- La única solución viable parece ser **numerar los paquetes**



¿Preguntas?

