Protocolos de Comunicación
Informe TPE Grupo 8

Primer cuatrimestre de 2022

Integrantes:

Negro, Juan Manuel	61225
Sambartolomeo, Mauro Daniel	61279
Morantes, Agustín Omar	61306

Fecha de entrega: Martes 21 de junio de 2022

Índice

Descripción de los protocolos y aplicaciones

Se implementó un servidor *SOCKS* siguiendo el *RFC 1928* capaz de manejar más de 500 conexiones concurrentes. Sin embargo, el servidor implementado no sigue todo lo pedido por el protocolo SOCKS según el RFC 1928, pues a decisión de la Cátedra no se implementó autenticación *GSSAPI*, y sólo se ofrece el comando *CONNECT*, pero no *BIND* ni *UDP ASSOCIATE*.

Además, se diseñó un protocolo binario basado en *TCP* para monitorear y configurar el servidor *SOCKS* mientras el mismo está en funcionamiento. Este protocolo se nombró *SHOES*, y su versión actual está definida y detallada en un archivo de texto en el proyecto, por lo que no se entrará en sumo detalle aquí para evitar la redundancia. El mismo servidor implementado para *SOCKS* también ofrece conexiones *SHOES* desde otro puerto (por defecto el 8080).

Como se tiene en mente en concepto de administración del *proxy*, también se implementó un sistema de registros. Cuando un cliente se conecte al *proxy*, se registrará por salida estándar información pertinente a la conexión (más detalle se puede encontrar en el manual del proyecto o en el *README* del mismo) que permitirá identificar lo sucedido en caso de algún evento importante. Por otro lado, también se implementó un sistema de disección de contraseñas donde el *proxy* registrará el usuario y contraseña ingresado por los clientes al intentar autenticarse en algún protocolo, actualmente sólo soportando *POP3*.

El protocolo exige a sus usuarios que se autentiquen mediante usuario y contraseña, pues la información que el servidor ofrece leer o modificar mediante el protocolo es sensitiva al funcionamiento del mismo, y se debe proteger de ingresos no autorizados.

La descripción del proyecto provista anteriormente es una general, no un detalle del mismo. Sin embargo, se encontrará más detalle de cada elemento y componente entre los archivos del proyecto. Para una descripción en detalle del proceso del servicio, se ofrece un archivo llamado *socks5d.8*, que aporta un listado y descripción de uso del servicio. Para una descripción detallada del diseño del protocolo *SHOES*, se ofrece un archivo de texto llamado *shoes-protocol.txt* que menciona cada característica del mismo en estilo *RFC*. Para una descripción detallada de tanto el comando del servidor como del cliente, se ofrece el archivo *README.md* que explica minuciosamente el uso de todo el proyecto, además de indicar una guía de instalación, parámetros aceptados del mismo, y precedencia de parámetros.

Problemas encontrados

Un problema se encontró es el de determinar el tamaño de los *buffers* de conexión. Por un lado, a mayor tamaño de *buffer* se logra un mayor *throughput* de datos, ya que se realizan menos copiados y por lo tanto menos *syscalls*. Por otro lado, se tienen 2 *buffers* por conexión, y si se reciben muchas conexiones juntas, sabiendo que el proxy puede manejar más de 500,

vamos a utilizar alrededor de 1000 *buffers*, por lo tanto tampoco puede ser excesivamente grande si queremos mantener un uso de memoria razonable. Finalmente, se decidió utilizar un buffer de 4 KB, ya que, asumiendo que llegan 500 conexiones simultáneas al *proxy*, se utilizarán menos de 4 MB de memoria, lo cual debería ser manejable por cualquier dispositivo moderno. De todas maneras, si el usuario lo desea, puede utilizar el protocolo *SHOES* para modificar el tamaño de *buffer* de nuevas conexiones en tiempo de ejecución para ajustar el *proxy* a sus necesidades.

Otro problema que se encontró es el de qué hacer con las conexiones entrantes cuando el servidor está atendiendo ya a su capacidad total. Una posible solución sería dejar que se encolen hasta que el *proxy* tenga espacio para poder aceptarlas. El problema es que ese tiempo puede ser muy grande, ya que no limitamos el tiempo que una conexión puede permanecer en uso. En ese caso se dejaría al cliente esperando por una cantidad de tiempo indeterminada. La solución que se decidió es la de rechazar las conexiones entrantes en dicho caso, guardando un *file descriptor* para poder aceptar e inmediatamente cerrar las conexiones entrantes cuando el *proxy* no tiene capacidad para atenderlas. De esta forma el cliente falla inmediatamente y no se lo hace esperar sin motivo.

Por otro lado, también se encontraron problemas en las lecturas o escrituras a los *buffers*. Como se debía realizar un servidor no bloqueante, luego no se puede determinar si la respuesta obtenida de parte de un cliente estaba completa o no, pues ciertos paquetes podrían no haber llegado. Luego, se debió procesar las respuestas *byte* a *byte*, para poder determinar cómo la información obtenida construía a la respuesta. Esto también significó incluir a la lectura de un paquete la información de por sí leída, lo cuál aumenta el espacio ocupado por una conexión al servidor.

Además, este también es el caso cuando se querían enviar respuestas a los clientes. Se debió utilizar el sistema de *buffers* circulares para poder mantener registro de qué información poseen los arreglos de datos y cuánto falta leer o escribir. De esta manera, se envía la mayor cantidad de datos posible y se mantiene un registro de cuánto más es necesario enviar, así solucionando el problema. Además, esta implementación permite definir tamaños de *buffers* de muy poco almacenamiento, por lo que las respuestas se enviarían en varias iteraciones. Esto haría que una conexión ocupe poco espacio en memoria, pero también ralentizará el funcionamiento del servidor.

Limitaciones

Una limitación de la implementación del servidor es que, por utilizar la *System Call Select*, entonces el sistema se limita a tener un máximo de 1024 *File Descriptors* registrados. Este factor limita la cantidad de conexiones concurrentes en el servidor, pues para poder tener una conexión con un cliente se necesita un *FD* para enviar mensajes hacia él. Sumando también 4 *FDs* para *sockets* pasivos, dos para *SOCKS* y otros dos para *SHOES*, para recibir conexiones tanto *IPv4* como *IPv6*, resulta en 1020 *FDs* para manejar conexiones. Además, para poder ser un *proxy SOCKS* se requiere otro *FD* para conectarnos al destino del cliente.

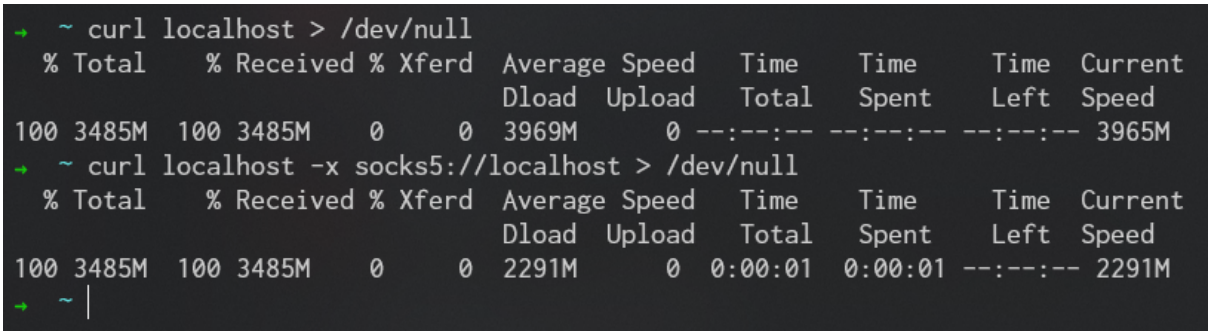
Luego, el límite a conexiones concurrentes es de 510 para *SOCKS* o 1020 para *SHOES*, por lo cuál se debe sumar la cantidad de *FDs* utilizados para saber si se puede aceptar una nueva conexión o no.

Por otro lado, en la sección de Problemas Encontrados se explicó cómo se reservó un *FD* para poder cerrar conexiones entrantes cuando no se espera tener la cantidad de *FDs* disponibles para atenderla. Luego, se podrían tener como máximo 509 conexiones *SOCKS* y una conexión *SHOES* o (excluyente) 1019 conexiones *SHOES*.

Dos limitaciones fundamentales para todo *proxy* son el *throughput* máximo del sistema y la memoria utilizada por el mismo. Siendo un *proxy*, el sistema debe copiar datos de una parte y transferirlos a la otra, siendo transparente a ambos extremos de la conexión. Este proceso de copiar datos entre *buffers* toma tiempo y degrada el *throughput* de la conexión relativo a si no se utiliza el *proxy*. Por otro lado, el uso de memoria del *proxy* debe ser el mínimo posible, siempre deseando que los sistemas sean lo más ligeros que se pueda.

La relación entre ambas variables se debe a que el tamaño de los *buffers* asignados para leer y escribir datos entre cliente y destino son fundamentales a ambas. Al aumentar el tamaño del *buffer*, el sistema ocupará más espacio en memoria pero también copiará datos a mayor velocidad. Esto se debe a varias razones, pero se explica en su mayoría pues recibir y enviar datos son *System Calls* que causan cambios de contexto (costosos en tiempo de procesamiento); y también se podrán leer y escribir más datos hacia las partes de la conexión en cada operación, aumentando el *throughput*. Al disminuir los tamaños de dichos *buffers*, el sistema ocupará menos espacio pero se ralentizará en consecuencia.

Para obtener el *throughput* máximo, se tomó el *buffer size* máximo posible según lo implementado, que con 2 *bytes* es de 65535 *bytes*, y se utilizó el *proxy* para descargar un archivo pesado (el archivo *ISO* de *Ubuntu 22.04* de más de 3.5 *Gb*) servido localmente utilizando *NGINX* para superar las velocidades de descarga de *Internet*. A continuación se ven ambas pruebas, con sus respectivos resultados:



Luego, se ve un *throughput* cercano a los 2300 Mb/*s*, y repetidas pruebas concluyeron similares resultados. Sin embargo, teniendo casi 64 *Kb* por *buffer* y necesitando dos para toda conexión *SOCKS*, se calcularía un total de 62.5 *Mb* de espacio ocupado solamente en tamaño de *buffers*. Aunque sería beneficioso para descargas de archivos pesados, las especificaciones del sistema utilizado para correr el *proxy* sería un factor limitante en la cantidad de memoria que el programa podría utilizar.

Por ende, se tomó la decisión de qué tamaño de *buffer* utilizar para conexiones del *proxy*. El valor final se definió en 4096 *bytes*. Utilizando estos valores y probando varias veces el servidor, se observaron velocidades aproximadas de 700 Mb/s, lo cuál es más que suficiente para conexiones *Internet*. Además, este tamaño resulta en un total de 4000 Kb de espacio ocupado en los *buffers* al tener 500 conexiones al *proxy*, lo cuál sería menor a 4 Mb y por ende se considera un valor razonable de memoria para un servidor que debe correr este programa.

Además, se tomó la decisión de definir un *buffer* para conexiones *SHOES* de 1024 *bytes*. Estos *buffers* no se utilizan normalmente para leer o escribir grandes cantidades de datos, siendo el paquete más grande el de listar usuarios en el caso de que existan 10 usuarios con nombres de usuario de máxima longitud resultando en 2560 *bytes* de *usernames*, resultando en 2818 *bytes* totales. Luego, la respuesta más larga posible se podría escribir en tan solo 3 escrituras al *buffer*, por lo que se considera aceptable.

Posibles extensiones

Tanto el *proxy SOCKS5* como el protocolo *SHOES* permiten extensiones. Se diseñaron e implementaron de forma tal que se puedan realizar abstracciones de los componentes fundamentales, y aún ser reemplazados por otros. Por ejemplo, el *proxy* maneja los *File Descriptors* de las conexiones utilizando funciones que abstraen la tarea de elegir el próximo *FD* que requiere una acción. Se utiliza la *System Call Select*, pero se podría mejorar la implementación utilizando otro mecanismo y reemplazar las funciones por otra de elección, pues para el *proxy* el manejo de *FDs* es transparente. Esto también es válido para el manejo de usuarios, donde se utilizan arreglos de punteros que apuntan a las estructuras que representan a los usuarios, pero se podría reemplazar por un archivo o por una base de datos avanzada, y el funcionamiento del *proxy* no cambiaría.

Por otro lado, también se puede extender el diseño del protocolo *SHOES*. Al utilizarse el concepto de familias de comandos, se podría extender el diseño para incorporar más familias que resuelvan otra categoría de problemas, pues se utilizan 2 de las 256 familias de comandos posibles. Esto también es cierto para los comandos en sí, donde se utilizan 5 comandos de los 256 posibles para la familia *PUT*, habiendo suficiente espacio para muchos comandos más. También vemos el mismo concepto en las respuestas, donde extensiones del protocolo podrían definir y utilizar respuestas particulares para sus comandos, habiendo otorgado dos valores de respuesta específicos al comando, además habiendo otros 250 valores sin utilizar.

Por ejemplo, se podría diseñar un comando que retorne el tiempo transcurrido sin interrupciones desde que el *proxy* comenzó a funcionar, asignando a la familia *GET* con el valor de *CMD* de 150, y definiendo parámetros para que la respuesta sea otorgada en la unidad de tiempo deseada. La flexibilidad de un protocolo binario permite codificar 256 valores distintos en un sólo *byte*, y esto es increíblemente útil a la hora enviar y recibir las respuestas codificadas.

Conclusiones

Se puede concluir, entonces, que se alcanzó un servidor apropiado para ser utilizado como *proxy SOCKS*. Navegar por *Internet* utilizando el *proxy* es algo transparente hasta para el usuario, pues si el proyecto se ejecuta en una computadora de igual capacidad a la de los integrantes de este equipo, se logrará una velocidad de aproximadamente 700 Mb/s o similar, lo cuál es más que suficiente considerando que el verdadero factor limitante a la conexión es la misma velocidad de descarga del cliente que probablemente menor a este valor.

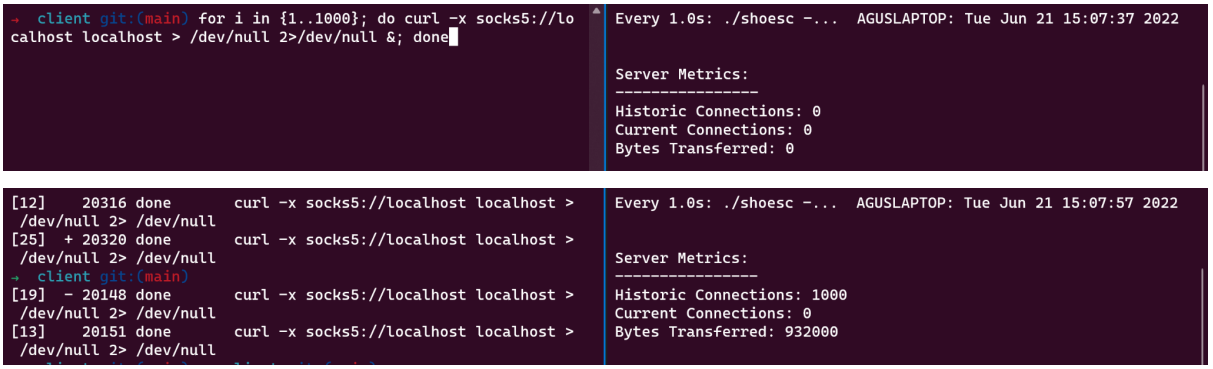
Se logró que el servidor sea no bloqueante, además logrando una concurrencia de como máximo 509 conexiones *SOCKS*, con todas ellas compartiendo el *throughput* registrado para que sus conexiones en *Internet* se mantengan estables y veloces. Además, cabe recordar cómo se podría extender el funcionamiento del servidor para no utilizar la *syscall Select* y poder utilizar un sistema más sofisticado, aumentando la cantidad máxima de usuarios.

Por otro lado, también se diseñó detalladamente un protocolo capaz de resolver todos los problemas de monitoreo y administración del servidor propuestos y de manera eficiente, aprovechando la flexibilidad de codificar acciones en el protocolo binario para ofrecer funcionamiento y amplia capacidad de extensión con una baja cantidad de espacio por paquete enviado.

Con el funcionamiento actual, con su rendimiento ya demostrado, y las considerables posibilidades para extender tanto protocolo como servidor para utilizar elementos más eficaces y avanzados, podemos concluir el proyecto y el análisis del mismo. Ante cualquier inconveniente o consulta, los autores estamos dispuestos a contestar cualquier pedido.

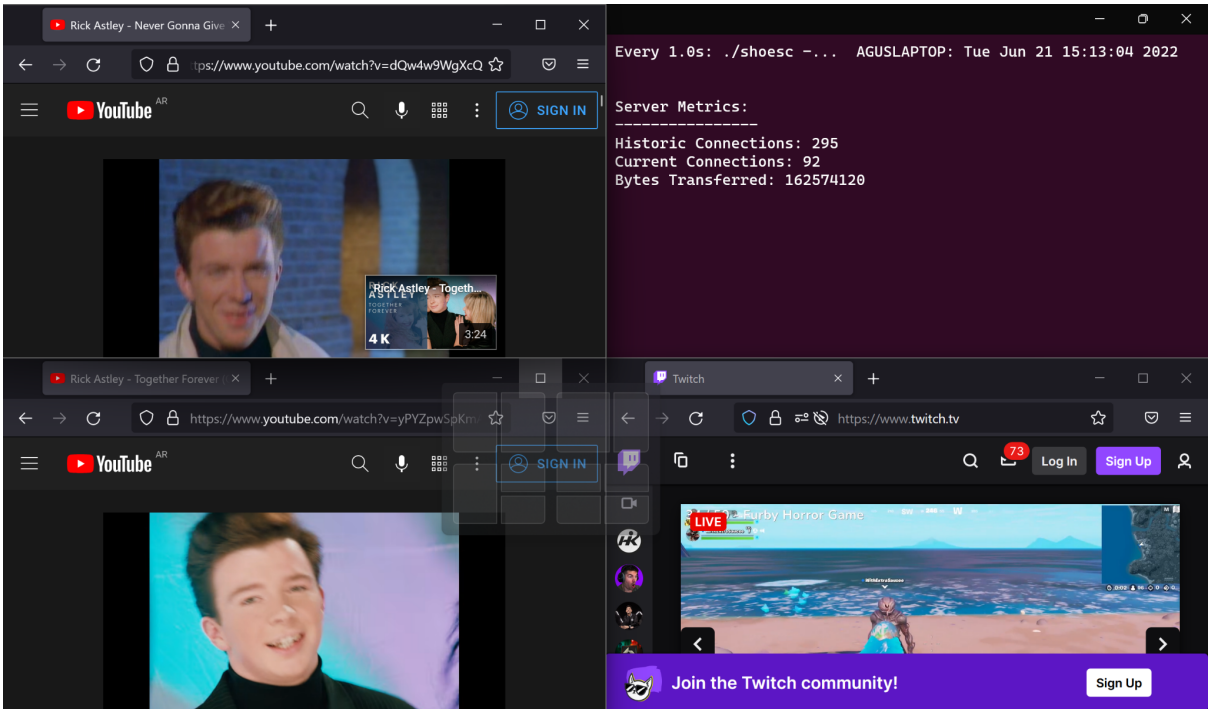
Ejemplos de prueba

En este ejemplo se ve cómo el servidor correctamente atiende 1000 conexiones a un servidor *http* local.



El *proxy* atiende las conexiones de manera rápida, por lo que nunca llegan a haber más de 500 conexiones simultáneas, en cuyo caso rechazaría las conexiones.

Aquí se puede ver *firefox* corriendo varios streams de video en simultáneo a través del *proxy*



En este ejemplo se puede ver el mismo *hash* calculado mediante sha256sum de la última iso de ubuntu servida desde un servidor local, tanto a través del proxy como sin el mismo, y por último el *hash* del archivo original servido

```
→  client git:(main) curl -s -x socks5://localhost localhost | sha256sum
b85286d9855f549ed9895763519f6a295a7698fb9c5c5345811b3eefadfb6f07  -
→  client git:(main) curl -s localhost | sha256sum
b85286d9855f549ed9895763519f6a295a7698fb9c5c5345811b3eefadfb6f07  -
→  client git:(main) sha256sum /var/www/html/ubuntu.iso
b85286d9855f549ed9895763519f6a295a7698fb9c5c5345811b3eefadfb6f07  /var/www/html/ubuntu.iso
→  client git:(main)
```

```
Every 1.0s: ./shoesc -...  AGUSLAPTOP: Tue Jun 21 15:22:23 2022


Server Metrics:
----------------
Historic Connections: 302
Current Connections: 0
Bytes Transferred: 3832756659
```

Guía de instalación

1. Primero, correr desde la carpeta raíz del proyecto el comando *make* para compilar tanto el *proxy* como el cliente de configuración.
2. Luego, los binarios se encontrarán en la subcarpeta *build* con los nombres *socks5d* para el *proxy* y *shoesc* para el cliente de *shoes*.
3. Ejecutar los binarios con los flags deseados. Con el flag *-h* puede ver todas las opciones posibles.

Instrucciones para la configuración

En primera instancia, para configurar el *proxy* uno debe indicar las opciones requeridas al correr el ejecutable con los flags indicados tanto en la *man page* provista como en la opción *-h* del binario.

En caso de haber indicado con la opción *-U* uno o más usuarios administradores, una vez el servidor esté corriendo puede acceder a más configuraciones mediante nuestro protocolo de configuración, *SHOES*.

Para hacer uso del protocolo *SHOES*, se provee una implementación, *shoesc*, cuyo binario se ubica en la carpeta *build* tras compilar el proyecto. Para utilizarlo, se debe llamar

pasando los flags de configuración deseados, también detallados al ejecutar ***shoesc -h*** o en el *README*.

El usuario de *shoesc* **DEBE** indicar el nombre de usuario y contraseña de un administrador (el cual es especificado al ejecutar al servidor *socks5*) con el flag *-u* para poder configurar el servidor. Notar que el *proxy* **NO** agrega ningún administrador por defecto por cuestiones de seguridad. En caso de que no se especifique un usuario administrador válido, *shoesc* no permitirá ningún tipo de cambio.

Ejemplos de configuración y monitoreo

Ejemplo de uso de la opción *-a* para agregar usuarios, la opción *-r* para removerlos y la opción *-g* para listarlos

```
→  client git:(main) ✗ ./shoesc -u shoes:shoes -a user:user -a user2:user2 -g

User 'user' added successfully

User 'user2' added successfully

USERS:
1: user
2: user2
→  client git:(main) ✗ ./shoesc -u shoes:shoes -a user3:user3 -r user -g

User 'user3' added successfully

User 'user' removed successfully

USERS:
1: user3
2: user2
```

Ejemplo de uso de la opción *-m* para ver las métricas del servidor

```
→  client git:(main) ✗ ./shoesc -u shoes:shoes -m

Server Metrics:
---------------
Historic Connections: 2
Current Connections: 0
Bytes Transferred: 848
```

Ejemplo de uso de la opción *-s* para modificar y obtener el estado actual del *pop3 spoofing*

```
➜  client git:(main) ✗ ./shoesc -u shoes:shoes -s

Spoof Status:
ON
➜  client git:(main) ✗ ./shoesc -u shoes:shoes -s0 -s

Spoofing status updated successfully

Spoof Status:
OFF
```

Ejemplo de modificación del *buffer* con la opción *-b*

```
➜  client git:(main) ✗ ./shoesc -u shoes:shoes -b 2048

Buffer size modified successfully
```

Decisiones de diseño del proyecto

Para la arquitectura del *proxy*, se hizo uso del *selector, buffer* y el motor de máquina de estados, ambos provistos por la cátedra, para facilitar la implementación del servidor, el cual realiza todas sus operaciones de forma no bloqueante en un único hilo de ejecución.

Al iniciar el *proxy* se leen y procesan los argumentos brindados y luego se crean dos *sockets pasivos* para conexiones *IPV4* e *IPV6* para clientes *socks5* y otros dos *sockets* para recibir conexiones *SHOES* y se registran al *selector*, para ser notificados de nuevas conexiones.

Al llegar una nueva conexión, se inicia una *state machine*, donde se atenderá todo el proceso de una conexión, desde la autenticación hasta *requests* y, en el caso de ser una conexión *socks5* el copiado de datos entre cliente y origen. Cuando ambos *hosts* cierran la conexión o en el caso de que ocurra algún error, se cierra la conexión y se libera la memoria utilizada.

En el caso del *parser* tanto de *SHOES* como de *socks5*, los *requests* se van leyendo y procesando *byte* a *byte* con otra *state machine* interna implementada manualmente, ya que, al ser no bloqueante y por cómo funciona *TCP*, no se puede recibir todo el *request* entero junto y es necesario ir procesando a medida que llegan los datos. La *state machine* simplifica este proceso.

Para realizar el copiado se guardan los datos recibidos en 2 *buffers* dependiendo del *host* del que se recibieron, y luego se envían hacia el destino correspondiente.