

Internetworking

Alejandro Furfaro

Setiembre 2017

Temario

1 Introducción

- Clasificación
- Componentes de un Sistema Distribuido

2 Conceptos de Networking

- Modelos de capas

3 Interfaz de sockets

- ¿socket? ¿Que es?
- Internet sockets
- Modelo Cliente Servidor
- API de nuestra librería de alto nivel
- API de sockets (POSIX)
- Hands On

1 Introducción

- Clasificación
- Componentes de un Sistema Distribuido

2 Conceptos de Networking

3 Interfaz de sockets

El Sistema Operativo: Abstracción al hardware

- 1 Los Sistemas operativos mediante la abstracción de procesos, nos independizan de los detalles relacionados con como cargar el programa en memoria, y establecen un API para utilizar Mecanismos de intercomunicación entre nuestros procesos.
- 2 Además esto nos asegura un funcionamiento independiente del hardware de base. Ej: Procesos escritos en C ejecutan sobre Linux de manera independiente en hardware Intel o ARM.

El Sistema Operativo: Abstracción al hardware

- ① Los Sistemas operativos mediante la abstracción de procesos, nos independizan de los detalles relacionados con como cargar el programa en memoria, y establecen un API para utilizar Mecanismos de intercomunicación entre nuestros procesos.
- ② Además esto nos asegura un funcionamiento independiente del hardware de base. Ej: Procesos escritos en C ejecutan sobre Linux de manera independiente en hardware Intel o ARM.

• El shared object que se encarga de cargar los procesos en Linux construye la imagen del proceso independientemente de la Arquitectura del hardware de base.

• Los procesos ejecutados en Linux tienen la misma vista de memoria.

• Los procesos ejecutados en Linux comparten la misma memoria.

• Los procesos ejecutados en Linux comparten la misma memoria.

• Los procesos ejecutados en Linux comparten la misma memoria.

El Sistema Operativo: Abstracción al hardware

- ① Los Sistemas operativos mediante la abstracción de procesos, nos independizan de los detalles relacionados con como cargar el programa en memoria, y establecen un API para utilizar Mecanismos de intercomunicación entre nuestros procesos.
- ② Además esto nos asegura un funcionamiento independiente del hardware de base. Ej: Procesos escritos en C ejecutan sobre Linux de manera independiente en hardware Intel o ARM.
 - El shared object que se encarga de cargar los procesos en Linux construye la imagen del proceso independientemente de la arquitectura del hardware de base.
 - Los comandos del shell que controlan y accionan sobre el proceso son idénticos.
 - El API para la creación y control de procesos es la misma
 - Los procesos se intercomunican mediante la misma interfaz de programación

El Sistema Operativo: Abstracción al hardware

- ① Los Sistemas operativos mediante la abstracción de procesos, nos independizan de los detalles relacionados con como cargar el programa en memoria, y establecen un API para utilizar Mecanismos de intercomunicación entre nuestros procesos.
- ② Además esto nos asegura un funcionamiento independiente del hardware de base. Ej: Procesos escritos en C ejecutan sobre Linux de manera independiente en hardware Intel o ARM.
 - El shared object que se encarga de cargar los procesos en Linux construye la imagen del proceso independientemente de la arquitectura del hardware de base.
 - Los comandos del shell que controlan y accionan sobre el proceso son idénticos.
 - El API para la creación y control de procesos es la misma
 - Los procesos se intercomunican mediante la misma interfaz de programación

El Sistema Operativo: Abstracción al hardware

- ① Los Sistemas operativos mediante la abstracción de procesos, nos independizan de los detalles relacionados con como cargar el programa en memoria, y establecen un API para utilizar Mecanismos de intercomunicación entre nuestros procesos.
- ② Además esto nos asegura un funcionamiento independiente del hardware de base. Ej: Procesos escritos en C ejecutan sobre Linux de manera independiente en hardware Intel o ARM.
 - El shared object que se encarga de cargar los procesos en Linux construye la imagen del proceso independientemente de la arquitectura del hardware de base.
 - Los comandos del shell que controlan y accionan sobre el proceso son idénticos.
 - El API para la creación y control de procesos es la misma
 - Los procesos se intercomunican mediante la misma interfaz de programación

El Sistema Operativo: Abstracción al hardware

- ① Los Sistemas operativos mediante la abstracción de procesos, nos independizan de los detalles relacionados con como cargar el programa en memoria, y establecen un API para utilizar Mecanismos de intercomunicación entre nuestros procesos.
- ② Además esto nos asegura un funcionamiento independiente del hardware de base. Ej: Procesos escritos en C ejecutan sobre Linux de manera independiente en hardware Intel o ARM.
 - El shared object que se encarga de cargar los procesos en Linux construye la imagen del proceso independientemente de la arquitectura del hardware de base.
 - Los comandos del shell que controlan y accionan sobre el proceso son idénticos.
 - El API para la creación y control de procesos es la misma
 - Los procesos se intercomunican mediante la misma interfaz de programación

El Sistema Operativo: Abstracción al hardware

- ① Los Sistemas operativos mediante la abstracción de procesos, nos independizan de los detalles relacionados con como cargar el programa en memoria, y establecen un API para utilizar Mecanismos de intercomunicación entre nuestros procesos.
- ② Además esto nos asegura un funcionamiento independiente del hardware de base. Ej: Procesos escritos en C ejecutan sobre Linux de manera independiente en hardware Intel o ARM.
 - El shared object que se encarga de cargar los procesos en Linux construye la imagen del proceso independientemente de la arquitectura del hardware de base.
 - Los comandos del shell que controlan y accionan sobre el proceso son idénticos.
 - El API para la creación y control de procesos es la misma
 - Los procesos se intercomunican mediante la misma interfaz de programación

¿Qué pasa en una red de datos?

- Desde la perspectiva del usuario, una red es un conjunto de aplicaciones distribuidas
- Detrás de esa vista hay dos elementos no visibles por el usuario

¿Que pasa en una red de datos?

- Desde la perspectiva del usuario, una red es un conjunto de aplicaciones distribuidas
- Detrás de esa vista hay dos elementos no visibles por el usuario
 - Una red física de interconexión compuesta por equipos de interconexión, equipos y medios de transmisión

¿Que pasa en una red de datos?

- Desde la perspectiva del usuario, una red es un conjunto de aplicaciones distribuidas
- Detrás de esa vista hay dos elementos no visibles por el usuario
 - 1 Una red física de interconexión compuesta por equipos de interconexión, equipos y medios de transmisión
 - 2 Una capa de software de mas alto nivel que el Sistema Operativo que permite aumentar el nivel de abstracción

¿Que pasa en una red de datos?

- Desde la perspectiva del usuario, una red es un conjunto de aplicaciones distribuidas
- Detrás de esa vista hay dos elementos no visibles por el usuario
 - 1 Una red física de interconexión compuesta por equipos de interconexión, equipos y medios de transmisión
 - 2 Una capa de software de mas alto nivel que el Sistema Operativo que permite aumentar el nivel de abstracción

¿Que pasa en una red de datos?

- Desde la perspectiva del usuario, una red es un conjunto de aplicaciones distribuidas
- Detrás de esa vista hay dos elementos no visibles por el usuario
 - ① Una red física de interconexión compuesta por equipos de interconexión, equipos y medios de transmisión
 - ② Una capa de software de mas alto nivel que el Sistema Operativo que permite aumentar el nivel de abstracción

intercomunicación de procesos distribuidos

Finalmente estamos buscando intercomunicar procesos

La diferencia es que ahora los procesos a intercomunicar residen en computadores diferentes, y pueden ejecutarse no solo en plataformas de hardware heterogéneas (Un main frame AS-400 accedido desde un iphone por ejemplo), sino que además los sistemas operativos son diferentes, u por lo tanto lo son las reglas para construir en memoria la imagen de un proceso, su identificación y los comandos para su control.

Se requieren reglas de mas alto nivel

El desafío es encontrar reglas que nos permitan independizarnos de los detalles ya no solo del hardware de base sino también del Sistema Operativo. Esto se logra en un sistema distribuido.

intercomunicación de procesos distribuidos

Finalmente estamos buscando intercomunicar procesos

La diferencia es que ahora los procesos a intercomunicar residen en computadores diferentes, y pueden ejecutarse no solo en plataformas de hardware heterogéneas (Un main frame AS-400 accedido desde un iphone por ejemplo), sino que además los sistemas operativos son diferentes, u por lo tanto lo son las reglas para construir en memoria la imagen de un proceso, su identificación y los comandos para su control.

Se requieren reglas de mas alto nivel

El desafío es encontrar reglas que nos permitan independizarnos de los detalles ya no solo del hardware de base sino también del Sistema Operativo. Esto se logra en un sistema distribuido.

1 Introducción

- Clasificación
- Componentes de un Sistema Distribuido

2 Conceptos de Networking

3 Interfaz de sockets

Enlaces



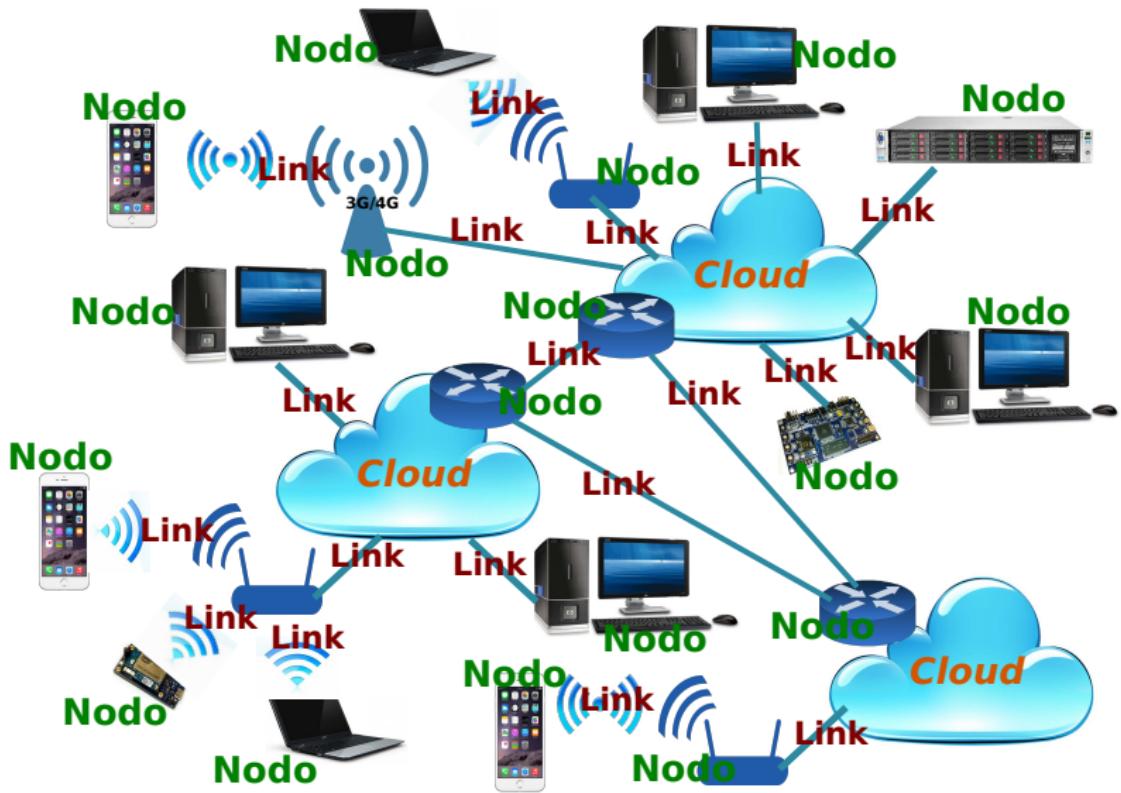
Nodos



La(s) Nube(s)



Todo junto interactuando



Encima de todo, las aplicaciones

- El usuario final no participa de toda esta complejidad
- Solo se preocupa de acceder a recursos desde aplicaciones de alto nivel. Ej: Con un navegador de internet ingresa a Amazon y consulta una gran base de datos en donde encuentra el artículo que busca.
- Todo esto sin preocuparse de detalles técnicos, tales como que motor de Base de datos usa Amazon, ni que arquitectura y sistema operativo tienen sus servidores.

1 Introducción

2 Conceptos de Networking

- Modelos de capas

3 Interfaz de sockets

Introducción

- Los sistemas de comunicaciones se representan mediante modelos conceptuales que caracterizan y estandarizan sus funciones independientemente de los detalles de tecnología.
- Su objetivo es proveer interoperabilidad de plataformas de hardware y software heterogéneas mediante la Definición de protocolos estándar
- La abstracción se logra cortando las funciones en capas horizontales (layers)
- Cada capa interactúa con sus capas vecinas a través de interfaces estándar.
- De este modo, todo sucede como si dos capas del mismo nivel interactúan con su par (peer) del otro extremo a través de una “conexión”.

Introducción

- Los sistemas de comunicaciones se representan mediante modelos conceptuales que caracterizan y estandarizan sus funciones independientemente de los detalles de tecnología.
- Su objetivo es proveer interoperabilidad de plataformas de hardware y software heterogéneas mediante la Definición de protocolos estándar
- La abstracción se logra cortando las funciones en capas horizontales (layers)
- Cada capa interactúa con sus capas vecinas a través de interfaces estándar.
- De este modo, todo sucede como si dos capas del mismo nivel interactúan con su par (peer) del otro extremo a través de una “conexión”.

Introducción

- Los sistemas de comunicaciones se representan mediante modelos conceptuales que caracterizan y estandarizan sus funciones independientemente de los detalles de tecnología.
- Su objetivo es proveer interoperabilidad de plataformas de hardware y software heterogéneas mediante la Definición de protocolos estándar
- La abstracción se logra cortando las funciones en capas horizontales (*layers*)
- Cada capa interactúa con sus capas vecinas a través de interfaces estándar.
- De este modo, todo sucede como si dos capas del mismo nivel interactúan con su par (peer) del otro extremo a través de una “conexión”.

Introducción

- Los sistemas de comunicaciones se representan mediante modelos conceptuales que caracterizan y estandarizan sus funciones independientemente de los detalles de tecnología.
- Su objetivo es proveer interoperabilidad de plataformas de hardware y software heterogéneas mediante la Definición de protocolos estándar
- La abstracción se logra cortando las funciones en capas horizontales (layers)
- Cada capa interactúa con sus capas vecinas a través de interfaces estándar.
- De este modo, todo sucede como si dos capas del mismo nivel interactúan con su par (peer) del otro extremo a través de una “conexión”.

Introducción

- Los sistemas de comunicaciones se representan mediante modelos conceptuales que caracterizan y estandarizan sus funciones independientemente de los detalles de tecnología.
- Su objetivo es proveer interoperabilidad de plataformas de hardware y software heterogéneas mediante la Definición de protocolos estándar
- La abstracción se logra cortando las funciones en capas horizontales (layers)
- Cada capa interactúa con sus capas vecinas a través de interfaces estándar.
- De este modo, todo sucede como si dos capas del mismo nivel interactúan con su par (peer) del otro extremo a través de una “conexión”.

Necesidad de estratificar

- Dividir en capas o layers horizontales las funciones de un sistema de comunicaciones, permite asignar a cada layer del modelo una función específica, y establecer que debe recibir para realizarla y que debe retornar una vez realizada.
- Además. de este modo cada capa dialoga de manera natural con su peer (par) en el extremo remoto, de manera independiente de las demás, ya que ambas desarrollan las mismas funciones.
- Finalmente dentro del nodo local, cada capa interactúa con sus adyacentes superior e inferior, pero solo actúa sobre la información que colocó su peer en el stack remoto

Necesidad de estratificar

- Dividir en capas o layers horizontales las funciones de un sistema de comunicaciones, permite asignar a cada layer del modelo una función específica, y establecer que debe recibir para realizarla y que debe retornar una vez realizada.
- Además, de este modo cada capa dialoga de manera natural con su peer (par) en el extremo remoto, de manera independiente de las demás, ya que ambas desarrollan las mismas funciones.
- Finalmente dentro del nodo local, cada capa interactúa con sus adyacentes superior e inferior, pero solo actúa sobre la información que colocó su peer en el stack remoto

Necesidad de estratificar

- Dividir en capas o layers horizontales las funciones de un sistema de comunicaciones, permite asignar a cada layer del modelo una función específica, y establecer que debe recibir para realizarla y que debe retornar una vez realizada.
- Además. de este modo cada capa dialoga de manera natural con su peer (par) en el extremo remoto, de manera independiente de las demás, ya que ambas desarrollan las mismas funciones.
- Finalmente dentro del nodo local, cada capa interactúa con sus adyacentes superior e inferior, pero solo actúa sobre la información que colocó su peer en el stack remoto

Necesidad de estratificar

- Dividir en capas o layers horizontales las funciones de un sistema de comunicaciones, permite asignar a cada layer del modelo una función específica, y establecer que debe recibir para realizarla y que debe retornar una vez realizada.
- Además. de este modo cada capa dialoga de manera natural con su peer (par) en el extremo remoto, de manera independiente de las demás, ya que ambas desarrollan las mismas funciones.
- Finalmente dentro del nodo local, cada capa interactúa con sus adyacentes superior e inferior, pero solo actúa sobre la información que colocó su peer en el stack remoto

Modelo OSI

Físico

Es el Nivel en el que las señales se acondicionan para viajar por el medio físico de transmisión (el cual no está incluido).

Modelo OSI

Enlace

Garantiza la transmisión de frames entre dos nodos conectados por el mismo medio físico, sin errores. Ej: IEEE 802.2, L2TP, LLDP, MAC, PPP

Físico

Es el Nivel en el que las señales se acondicionan para viajar por el medio físico de transmisión (el cual no está incluido).

Modelo OSI

Red	Transmisión de paquetes entre nodos en una internet, direccionamiento único, control y enrutamiento. Ej: IPv4, IPv6, IPSEC, AppleTalk
Enlace	Garantiza la transmisión de frames entre dos nodos conectados por el mismo medio físico, sin errores. Ej: IEEE 802.2, L2TP, LLDP, MAC, PPP
Físico	Es el Nivel en el que las señales se acondicionan para viajar por el medio físico de transmisión (el cual no está incluido).

Modelo OSI

Transporte	Transmisión confiable de segmentos entre puntos de una red, multiplexado y acuse de recibo. Ej: TCP, UDP, MBF
Red	Transmisión de paquetes entre nodos en una internet, direccionamiento único, control y enrutamiento. Ej: IPv4, IPv6, IPSEC, AppleTalk
Enlace	Garantiza la transmisión de frames entre dos nodos conectados por el mismo medio físico, sin errores. Ej: IEEE 802.2, L2TP, LLDP, MAC, PPP
Físico	Es el Nivel en el que las señales se acondicionan para viajar por el medio físico de transmisión (el cual no está incluido).

Modelo OSI

Sesión	Establecimiento de sesiones entre nodos para transmisión bidireccional. Ej: HTTP, HTTPS, SSH, FTP, SFTP, NFS
Transporte	Transmisión confiable de segmentos entre puntos de una red, multiplexado y acuse de recibo. Ej: TCP, UDP, MBF
Red	Transmisión de paquetes entre nodos en una internet, direccionamiento único, control y enrutamiento. Ej: IPv4, IPv6, IPSEC, AppleTalk
Enlace	Garantiza la transmisión de frames entre dos nodos conectados por el mismo medio físico, sin errores. Ej: IEEE 802.2, L2TP, LLDP, MAC, PPP
Físico	Es el Nivel en el que las señales se acondicionan para viajar por el medio físico de transmisión (el cual no está incluido).

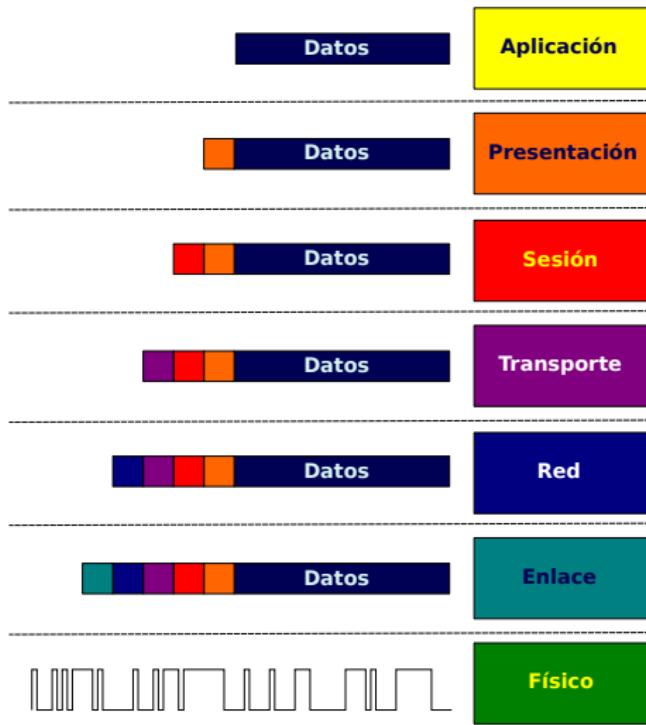
Modelo OSI

Presentación	Acondicionamiento de la información hacia o desde la aplicación. Encriptado/Desencriptado, Compresión/Descompresión, etc
Sesión	Establecimiento de sesiones entre nodos para transmisión bidireccional. Ej: HTTP, HTTPS, SSH, FTP, SFTP, NFS
Transporte	Transmisión confiable de segmentos entre puntos de una red, multiplexado y acuse de recibo. Ej: TCP, UDP, MBF
Red	Transmisión de paquetes entre nodos en una internet, direccionamiento único, control y enrutamiento. Ej: IPv4, IPv6, IPSEC, AppleTalk
Enlace	Garantiza la transmisión de frames entre dos nodos conectados por el mismo medio físico, sin errores. Ej: IEEE 802.2, L2TP, LLDP, MAC, PPP
Físico	Es el Nivel en el que las señales se acondicionan para viajar por el medio físico de transmisión (el cual no está incluido).

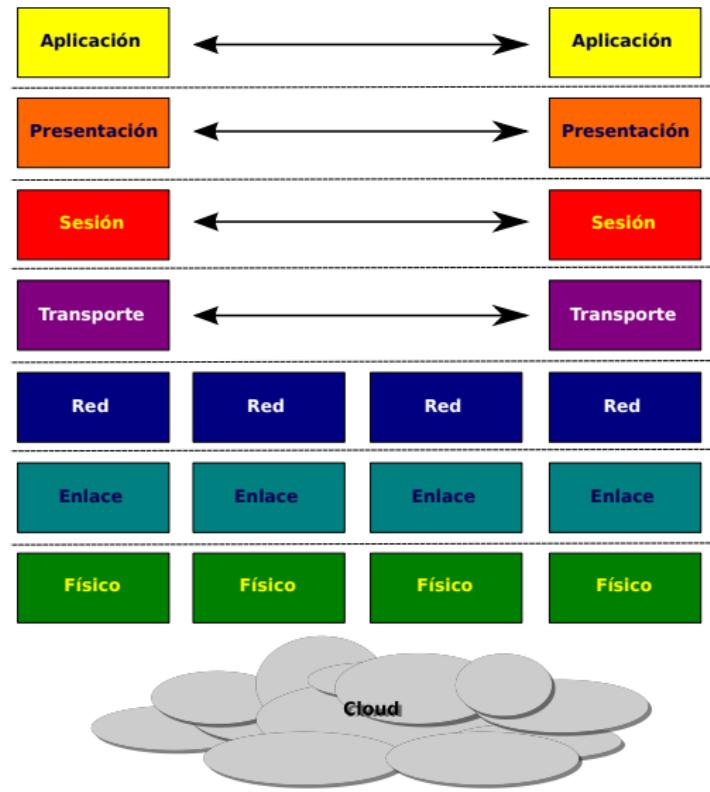
Modelo OSI

Aplicación	Interfaz con el usuario del mas alto nivel. Maquinas virtuales, APIs, Acceso a servicios.
Presentación	Acondicionamiento de la información hacia o desde la aplicación. Encriptado/Desencriptado, Compresión/Descompresión, etc
Sesión	Establecimiento de sesiones entre nodos para transmisión bidireccional. Ej: HTTP, HTTPS, SSH, FTP, SFTP, NFS
Transporte	Transmisión confiable de segmentos entre puntos de una red, multiplexado y acuse de recibo. Ej: TCP, UDP, MBF
Red	Transmisión de paquetes entre nodos en una internet, direccionamiento único, control y enrutamiento. Ej: IPv4, IPv6, IPSEC, AppleTalk
Enlace	Garantiza la transmisión de frames entre dos nodos conectados por el mismo medio físico, sin errores. Ej: IEEE 802.2, L2TP, LLDP, MAC, PPP
Físico	Es el Nivel en el que las señales se acondicionan para viajar por el medio físico de transmisión (el cual no está incluido).

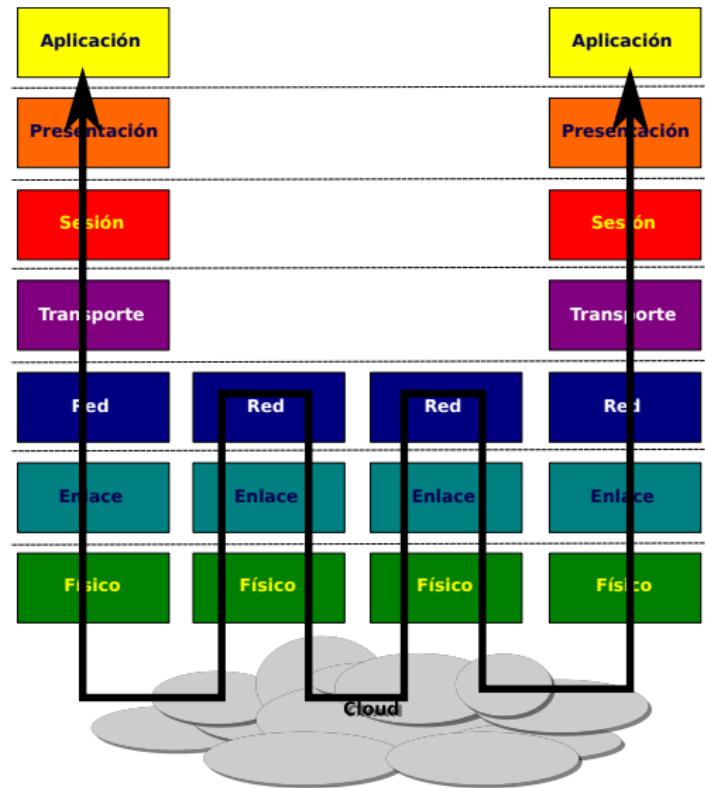
Modelo OSI



Modelo OSI, virtual



Modelo OSI, real



Modelo DARPA TCP/IP

Aplicación

Presentación

Sesión

Transporte

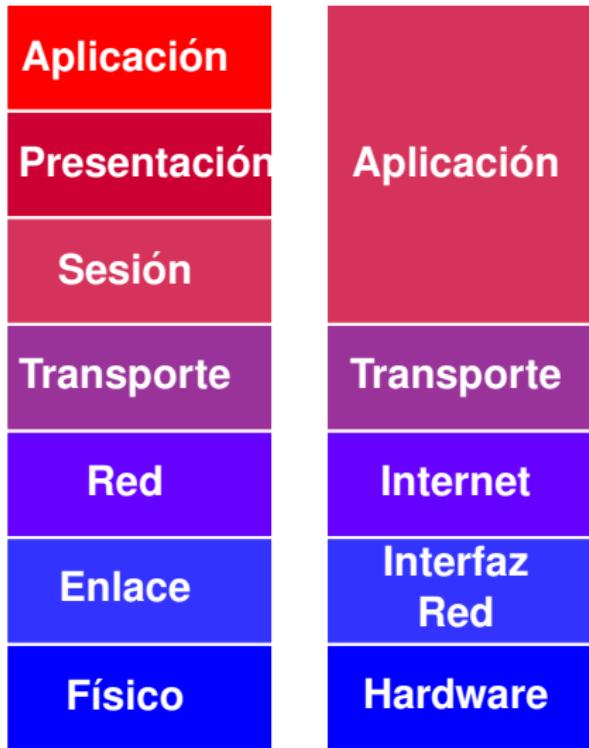
Red

Enlace

Físico

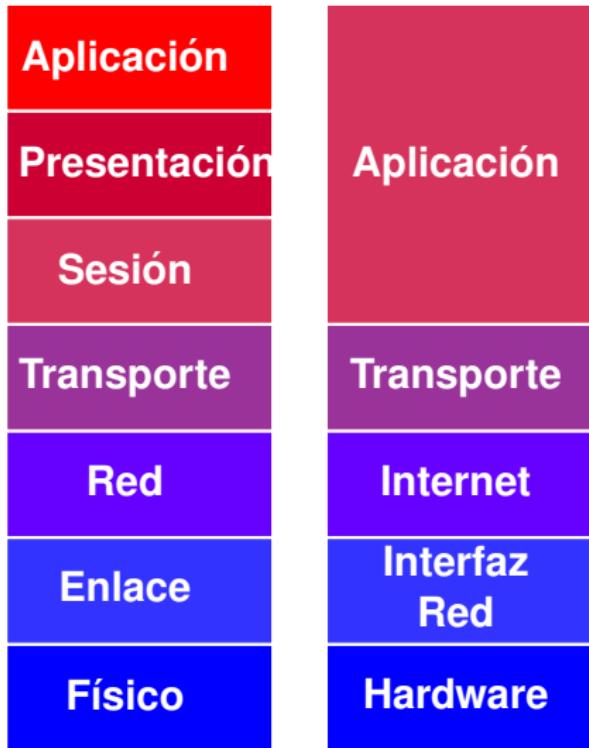
- Ambos modelos coinciden en las cuatro capas inferiores. Difieren en ocasiones en los nombres de las mismas pero tienen la misma función cada una
- La diferencia es que las tres capas superiores del modelo OSI, DARPA las concentra en una única capa.
- Nuestro ámbito será DARPA TCP/IP

Modelo DARPA TCP/IP



- Ambos modelos coinciden en las cuatro capas inferiores. Difieren en ocasiones en los nombres de las mismas pero tienen la misma función cada una
- La diferencia es que las tres capas superiores del modelo OSI, DARPA las concentra en una única capa.
- Nuestro ámbito será DARPA TCP/IP

Modelo DARPA TCP/IP



- Ambos modelos coinciden en las cuatro capas inferiores. Difieren en ocasiones en los nombres de las mismas pero tienen la misma función cada una
- La diferencia es que las tres capas superiores del modelo OSI, DARPA las concentra en una única capa.
- Nuestro ámbito será DARPA TCP/IP

Modelo DARPA TCP/IP



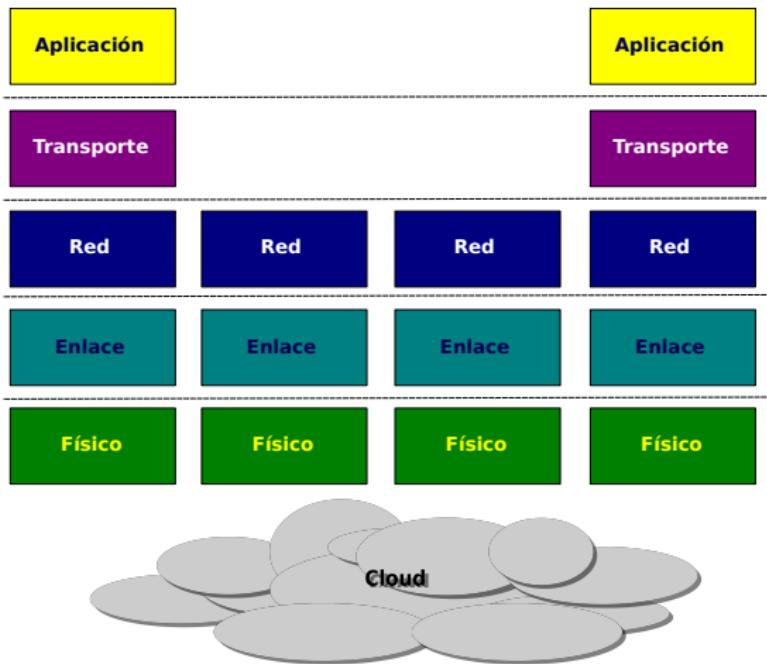
- Ambos modelos coinciden en las cuatro capas inferiores. Difieren en ocasiones en los nombres de las mismas pero tienen la misma función cada una
- La diferencia es que las tres capas superiores del modelo OSI, DARPA las concentra en una única capa.
- Nuestro ámbito será DARPA TCP/IP

Modelo DARPA TCP/IP



- Ambos modelos coinciden en las cuatro capas inferiores. Difieren en ocasiones en los nombres de las mismas pero tienen la misma función cada una
- La diferencia es que las tres capas superiores del modelo OSI, DARPA las concentra en una única capa.
- Nuestro ámbito será DARPA TCP/IP

Modelo DARPA



Encapsulado de paquetes

- Cada capa del modelo maneja un protocolo
- El protocolo de las capas depende del sistema que se utilice
- Cada capa incluye un conjunto de datos de control llamado **Header** en el que se manifiestan todos los parámetros necesarios para que cada extremo interprete el protocolo y de ese modo se pueda lograr la mencionada “conexión” virtual entre los extremos a nivel de cada capa
- De este modo a medida que baja por las diferentes capas se van agregando encabezados.

Encapsulado de paquetes

- Cada capa del modelo maneja un protocolo
- El protocolo de las capas depende del sistema que se utilice
- Cada capa incluye un conjunto de datos de control llamado **Header** en el que se manifiestan todos los parámetros necesarios para que cada extremo interprete el protocolo y de ese modo se pueda lograr la mencionada “conexión” virtual entre los extremos a nivel de cada capa
- De este modo a medida que baja por las diferentes capas se van agregando encabezados.

Encapsulado de paquetes

- Cada capa del modelo maneja un protocolo
- El protocolo de las capas depende del sistema que se utilice
- Cada capa incluye un conjunto de datos de control llamado **Header** en el que se manifiestan todos los parámetros necesarios para que cada extremo interprete el protocolo y de ese modo se pueda lograr la mencionada “conexión” virtual entre los extremos a nivel de cada capa
- De este modo a medida que baja por las diferentes capas se van agregando encabezados.

Encapsulado de paquetes

- Cada capa del modelo maneja un protocolo
- El protocolo de las capas depende del sistema que se utilice
- Cada capa incluye un conjunto de datos de control llamado **Header** en el que se manifiestan todos los parámetros necesarios para que cada extremo interprete el protocolo y de ese modo se pueda lograr la mencionada “conexión” virtual entre los extremos a nivel de cada capa
- De este modo a medida que baja por las diferentes capas se van agregando encabezados.

Encapsulado de paquetes

- Cada capa del modelo maneja un protocolo
- El protocolo de las capas depende del sistema que se utilice
- Cada capa incluye un conjunto de datos de control llamado **Header** en el que se manifiestan todos los parámetros necesarios para que cada extremo interprete el protocolo y de ese modo se pueda lograr la mencionada “conexión” virtual entre los extremos a nivel de cada capa
- De este modo a medida que baja por las diferentes capas se van agregando encabezados.

Ejemplo



1 Introducción

2 Conceptos de Networking

3 Interfaz de sockets

- ¿socket? ¿Que es?
- Internet sockets
- Modelo Cliente Servidor
- API de nuestra librería de alto nivel
- API de sockets (POSIX)
- Hands On

Definición

socket

Es un canal de comunicación bidireccional entre dos procesos que puede manejarse mediante un simple file descriptor

- Parece no ser gran cosa, ya que dicho de este modo es como un **Named FIFO**, pero con la ventaja de ser bidireccional.
- Sin embargo los **socket** nos permiten intercomunicar procesos que se ejecuten dentro del mismo computador, o ... ¡En computadores diferentes!
- Si bien en principio la interfaz se diseñó para UNIX, actualmente todos los sistemas operativos (¡hasta Windows !) lo tienen implementado.
- Entonces podemos intercomunicar procesos que ejecutan en computadores diferentes, cuya arquitectura y sistema operativo es heterogénea.

Definición

socket

Es un canal de comunicación bidireccional entre dos procesos que puede manejarse mediante un simple file descriptor

- Parece no ser gran cosa, ya que dicho de este modo es como un **Named FIFO**, pero con la ventaja de ser bidireccional.
- Sin embargo los **socket** nos permiten intercomunicar procesos que se ejecuten dentro del mismo computador, o ... ¡En computadores diferentes!
- Si bien en principio la interfaz se diseñó para UNIX, actualmente todos los sistemas operativos (¡hasta Windows !) lo tienen implementado.
- Entonces podemos intercomunicar procesos que ejecutan en computadores diferentes, cuya arquitectura y sistema operativo es heterogénea.

Definición

socket

Es un canal de comunicación bidireccional entre dos procesos que puede manejarse mediante un simple file descriptor

- Parece no ser gran cosa, ya que dicho de este modo es como un **Named FIFO**, pero con la ventaja de ser bidireccional.
- Sin embargo los **socket** nos permiten intercomunicar procesos que se ejecuten dentro del mismo computador, o . . . ¡En computadores diferentes!
- Si bien en principio la interfaz se diseñó para UNIX, actualmente todos los sistemas operativos (¡hasta Windows !) lo tienen implementado.
- Entonces podemos intercomunicar procesos que ejecutan en computadores diferentes, cuya arquitectura y sistema operativo es heterogénea.

Definición

socket

Es un canal de comunicación bidireccional entre dos procesos que puede manejarse mediante un simple file descriptor

- Parece no ser gran cosa, ya que dicho de este modo es como un **Named FIFO**, pero con la ventaja de ser bidireccional.
- Sin embargo los **socket** nos permiten intercomunicar procesos que se ejecuten dentro del mismo computador, o . . . ¡En computadores diferentes!
- Si bien en principio la interfaz se diseñó para UNIX, actualmente todos los sistemas operativos (¡hasta Windows !) lo tienen implementado.
- Entonces podemos intercomunicar procesos que ejecutan en computadores diferentes, cuya arquitectura y sistema operativo es heterogénea.

Definición

socket

Es un canal de comunicación bidireccional entre dos procesos que puede manejarse mediante un simple file descriptor

- Parece no ser gran cosa, ya que dicho de este modo es como un **Named FIFO**, pero con la ventaja de ser bidireccional.
- Sin embargo los **socket** nos permiten intercomunicar procesos que se ejecuten dentro del mismo computador, o . . . ¡En computadores diferentes!
- Si bien en principio la interfaz se diseñó para UNIX, actualmente todos los sistemas operativos (¡hasta Windows !) lo tienen implementado.
- Entonces podemos intercomunicar procesos que ejecutan en computadores diferentes, cuya arquitectura y sistema operativo es heterogénea.

POSIX... una vez mas

- Si un **socket** se maneja mediante un file descriptor, se pueden utilizar para recibir y enviar información por la red, las funciones **read ()** y **write ()**.
- Recordemos el lema que caracteriza a los sistemas UNIX: "***everything is a file***". Como vemos aplica hasta para las conexiones de red.
- Por supuesto la interfaz de programación de sockets posee funciones específicas que proveen mayor control sobre las transacciones, pero la versatilidad de la filosofía de tratar hasta las conexiones de red como si fuesen un simple archivo es digna de destacar.

POSIX... una vez mas

- Si un **socket** se maneja mediante un file descriptor, se pueden utilizar para recibir y enviar información por la red, las funciones **read ()** y **write ()**.
- Recordemos el lema que caracteriza a los sistemas UNIX: “***everything is a file***”. Como vemos aplica hasta para las conexiones de red.
- Por supuesto la interfaz de programación de sockets posee funciones específicas que proveen mayor control sobre las transacciones, pero la versatilidad de la filosofía de tratar hasta las conexiones de red como si fuesen un simple archivo es digna de destacar.

POSIX... una vez mas

- Si un **socket** se maneja mediante un file descriptor, se pueden utilizar para recibir y enviar información por la red, las funciones **read ()** y **write ()**.
- Recordemos el lema que caracteriza a los sistemas UNIX: “***everything is a file***”. Como vemos aplica hasta para las conexiones de red.
- Por supuesto la interfaz de programación de sockets posee funciones específicas que proveen mayor control sobre las transacciones, pero la versatilidad de la filosofía de tratar hasta las conexiones de red como si fuesen un simple archivo es digna de destacar.

Diferentes tipos de sockets

- Internet sockets: Son los que manejan las direcciones de internet definidas por DARPA
- Unix sockets: Intercomunican procesos dentro de un sistema operativo Unix y se manifiestan como un nodo local en el file system.
- X25 sockets: definidos por el CCITT en su momento tuvieron cierta vigencia, pero este tipo de redes ya no están vigentes.
- En general podemos encontrar una variedad de sockets.

Diferentes tipos de sockets

- Internet sockets: Son los que manejan las direcciones de internet definidas por DARPA
- Unix sockets: Intercomunican procesos dentro de un sistema operativo Unix y se manifiestan como un nodo local en el file system.
- X25 sockets: definidos por el CCITT en su momento tuvieron cierta vigencia, pero este tipo de redes ya no están vigentes.
- En general podemos encontrar una variedad de sockets.

Diferentes tipos de sockets

- Internet sockets: Son los que manejan las direcciones de internet definidas por DARPA
- Unix sockets: Intercomunican procesos dentro de un sistema operativo Unix y se manifiestan como un nodo local en el file system.
- X25 sockets: definidos por el CCITT en su momento tuvieron cierta vigencia, pero este tipo de redes ya no están vigentes.
- En general podemos encontrar una variedad de sockets.

Diferentes tipos de sockets

- Internet sockets: Son los que manejan las direcciones de internet definidas por DARPA
- Unix sockets: Intercomunican procesos dentro de un sistema operativo Unix y se manifiestan como un nodo local en el file system.
- X25 sockets: definidos por el CCITT en su momento tuvieron cierta vigencia, pero este tipo de redes ya no están vigentes.
- En general podemos encontrar una variedad de sockets.

Diferentes tipos de sockets

Scope de nuestro estudio

- Internet sockets: Son los que manejan las direcciones de internet definidas por DARPA
- Unix sockets: Intercomunican procesos dentro de un sistema operativo Unix y se manifiestan como un nodo local en el file system.
- X25 sockets: definidos por el CCITT en su momento tuvieron cierta vigencia, pero este tipo de redes ya no están vigentes.
- En general podemos encontrar una variedad de sockets.

1 Introducción

2 Conceptos de Networking

3 Interfaz de sockets

- ¿socket? ¿Que es?
- **Internet sockets**
- Modelo Cliente Servidor
- API de nuestra librería de alto nivel
- API de sockets (POSIX)
- Hands On

Tipos de Internet sockets

Nuevamente tenemos varios tipos de sockets

- **SOCK_DGRAM:** Son sockets para comunicaciones en modo no conectado, con envío de datagramas de tamaño limitado.
- **SOCK_STREAM:** Para comunicaciones confiables en modo conectado, de dos vías y con tamaño variable de los mensajes de datos.
- **SOCK_RAW:** permite el acceso a protocolos de más bajo nivel como el IP (nivel de red)
- **SOCK_SEQPACKET:** tiene las características del SOCK_STREAM pero además el tamaño de los mensajes es fijo.

Tipos de Internet sockets

Nuevamente tenemos varios tipos de sockets

- **SOCK_DGRAM:** Son sockets para comunicaciones en modo no conectado, con envío de datagramas de tamaño limitado.
- **SOCK_STREAM:** Para comunicaciones confiables en modo conectado, de dos vías y con tamaño variable de los mensajes de datos.
- **SOCK_RAW:** permite el acceso a protocolos de más bajo nivel como el IP (nivel de red)
- **SOCK_SEQPACKET:** tiene las características del SOCK_STREAM pero además el tamaño de los mensajes es fijo.

Tipos de Internet sockets

Nuevamente tenemos varios tipos de sockets

- **SOCK_DGRAM**: Son sockets para comunicaciones en modo no conectado, con envío de datagramas de tamaño limitado.
- **SOCK_STREAM**: Para comunicaciones confiables en modo conectado, de dos vías y con tamaño variable de los mensajes de datos.
- **SOCK_RAW**: permite el acceso a protocolos de más bajo nivel como el IP (nivel de red)
- **SOCK_SEQPACKET**: tiene las características del SOCK_STREAM pero además el tamaño de los mensajes es fijo.

Tipos de Internet sockets

Nuevamente tenemos varios tipos de sockets

- **SOCK_DGRAM**: Son sockets para comunicaciones en modo no conectado, con envío de datagramas de tamaño limitado.
- **SOCK_STREAM**: Para comunicaciones confiables en modo conectado, de dos vías y con tamaño variable de los mensajes de datos.
- **SOCK_RAW**: permite el acceso a protocolos de más bajo nivel como el IP (nivel de red)
- **SOCK_SEQPACKET**: tiene las características del SOCK_STREAM pero además el tamaño de los mensajes es fijo.

Tipos de Internet sockets

Nuevamente tenemos varios tipos de sockets

Scope de nuestro estudio

- **SOCK_DGRAM**: Son sockets para comunicaciones en modo no conectado, con envío de datagramas de tamaño limitado.
- **SOCK_STREAM**: Para comunicaciones confiables en modo conectado, de dos vías y con tamaño variable de los mensajes de datos.
- **SOCK_RAW**: permite el acceso a protocolos de más bajo nivel como el IP (nivel de red)
- **SOCK_SEQPACKET**: tiene las características del SOCK_STREAM pero además el tamaño de los mensajes es fijo.

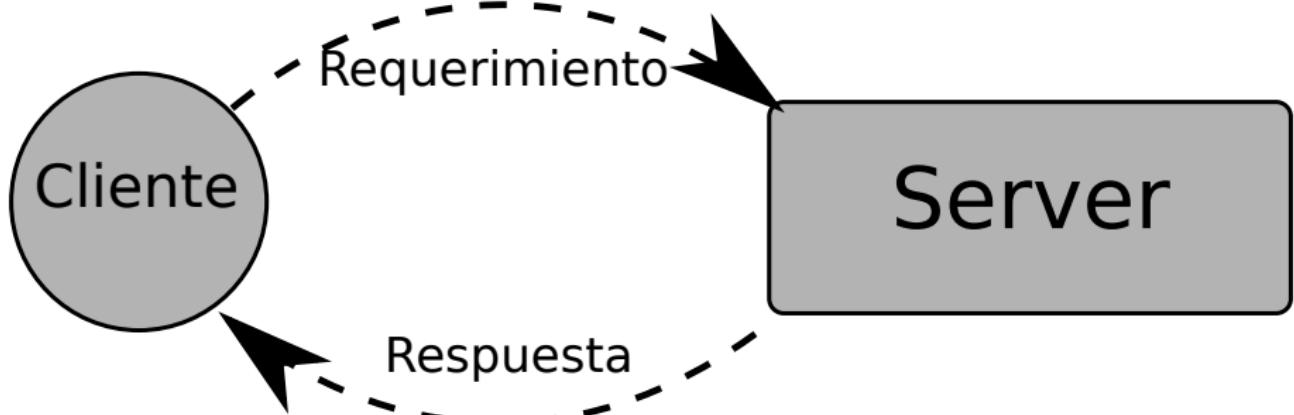
1 Introducción

2 Conceptos de Networking

3 Interfaz de sockets

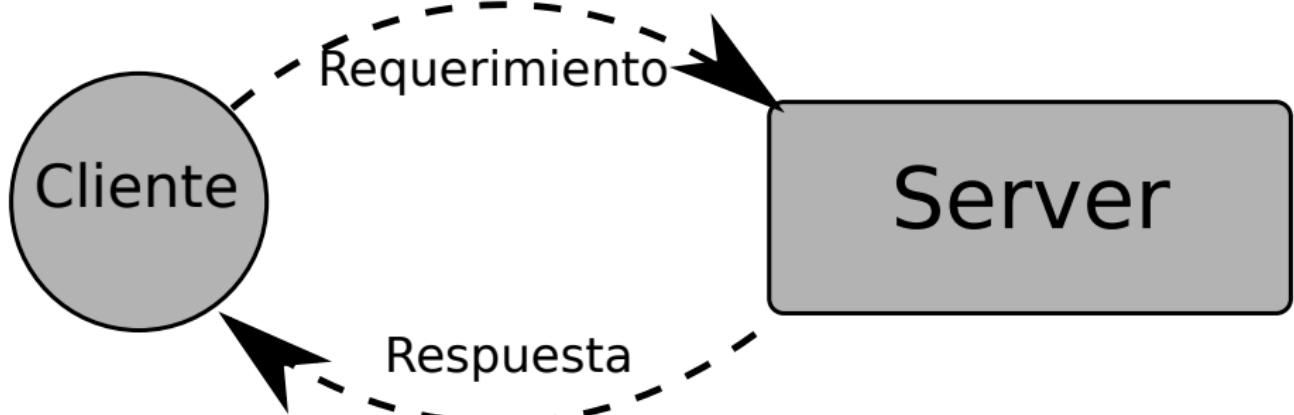
- ¿socket? ¿Que es?
- Internet sockets
- **Modelo Cliente Servidor**
- API de nuestra librería de alto nivel
- API de sockets (POSIX)
- Hands On

Modelo cliente servidor



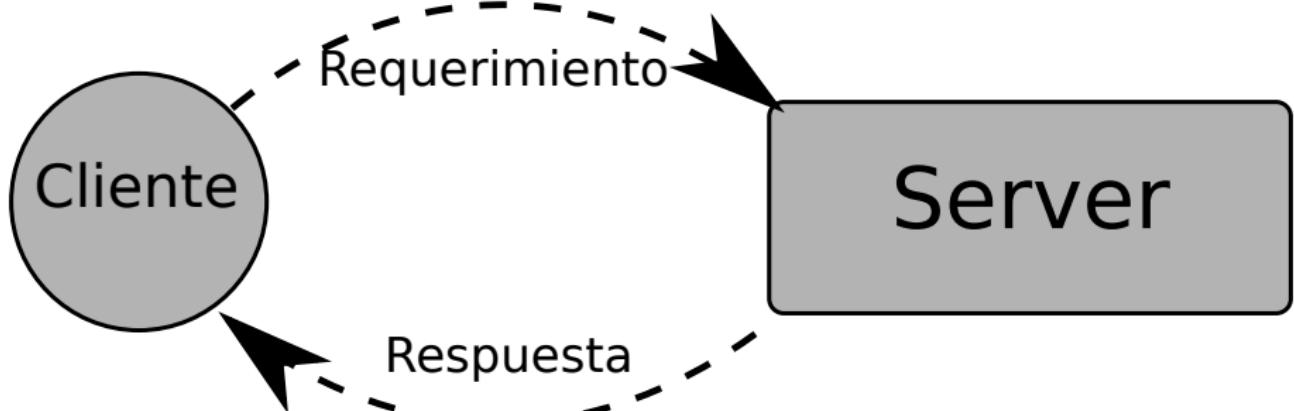
- Cliente y Servidor son dos procesos.
- Pueden estar en el mismo computador o en computadores diferentes pero en tal caso los computadores deben estar conectados por una red de datos
- La idea es distribuir el procesamiento en forma colaborativa.

Modelo cliente servidor



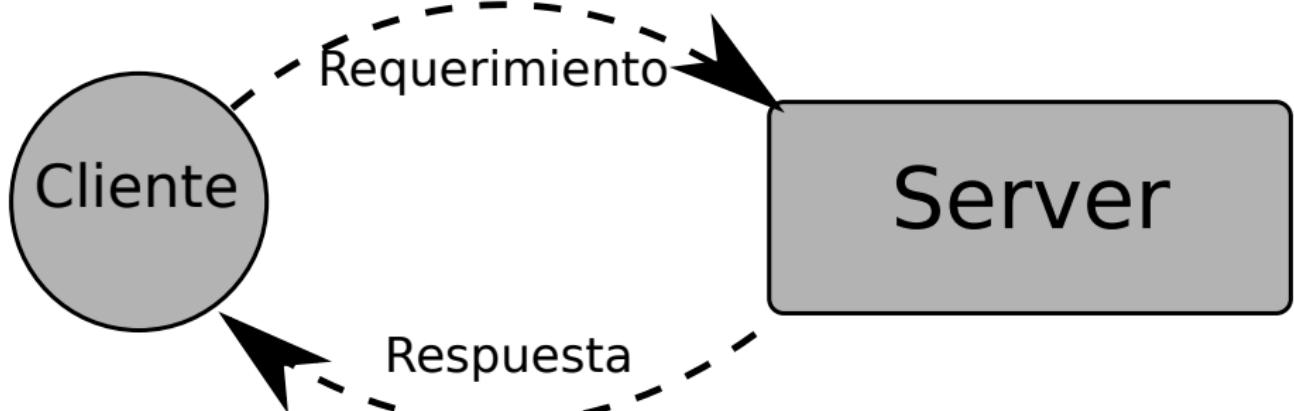
- Cliente y Servidor son dos procesos.
- Pueden estar en el mismo computador o en computadores diferentes pero en tal caso los computadores deben estar conectados por una red de datos
- La idea es distribuir el procesamiento en forma colaborativa.

Modelo cliente servidor



- Cliente y Servidor son dos procesos.
- Pueden estar en el mismo computador o en computadores diferentes pero en tal caso los computadores deben estar conectados por una red de datos
- La idea es distribuir el procesamiento en forma colaborativa.

Modelo cliente servidor



- Cliente y Servidor son dos procesos.
- Pueden estar en el mismo computador o en computadores diferentes pero en tal caso los computadores deben estar conectados por una red de datos
- La idea es distribuir el procesamiento en forma colaborativa.

Caso general: Intercomunicación por red

- En este caso el cliente y el servidor son procesos que ejecutan siempre en computadores diferentes.
- No obstante la red puede ser simulada y ambos procesos ejecutar en el mismo computador como si la red estuviese en medio.
- Los computadores pueden ser de diferente hardware (Intel uno y ARM el otro), y también pueden tener sistemas operativos diferentes (MAC uno y LINUX el otro).
- Sin embargo esto detalles quedan completamente en segundo plano.
- Los sistemas Operativos de los computadores ni siquiera necesitan ser POSIX ya que la interfaz de sockets es de mas alto nivel aun
- Sin embargo en POSIX el socket es tratado como un archivo, de modo que solo es necesario que si ambos sistemas son POSIX compatibles, no solo nos sirven los mismos fuentes sino que además podemos manejarlo muy fácilmente.

Caso general: Intercomunicación por red

- En este caso el cliente y el servidor son procesos que ejecutan siempre en computadores diferentes.
- No obstante la red puede ser simulada y ambos procesos ejecutar en el mismo computador como si la red estuviese en medio.
- Los computadores pueden ser de diferente hardware (Intel uno y ARM el otro), y también pueden tener sistemas operativos diferentes (MAC uno y LINUX el otro).
- Sin embargo esto detalles quedan completamente en segundo plano.
- Los sistemas Operativos de los computadores ni siquiera necesitan ser POSIX ya que la interfaz de sockets es de mas alto nivel aun
- Sin embargo en POSIX el socket es tratado como un archivo, de modo que solo es necesario que si ambos sistemas son POSIX compatibles, no solo nos sirven los mismos fuentes sino que además podemos manejarlo muy fácilmente.

Caso general: Intercomunicación por red

- En este caso el cliente y el servidor son procesos que ejecutan siempre en computadores diferentes.
- No obstante la red puede ser simulada y ambos procesos ejecutar en el mismo computador como si la red estuviese en medio.
- Los computadores pueden ser de diferente hardware (Intel uno y ARM el otro), y también pueden tener sistemas operativos diferentes (MAC uno y LINUX el otro).
- Sin embargo esto detalles quedan completamente en segundo plano.
- Los sistemas Operativos de los computadores ni siquiera necesitan ser POSIX ya que la interfaz de sockets es de mas alto nivel aun
- Sin embargo en POSIX el socket es tratado como un archivo, de modo que solo es necesario que si ambos sistemas son POSIX compatibles, no solo nos sirven los mismos fuentes sino que además podemos manejarlo muy fácilmente.

Caso general: Intercomunicación por red

- En este caso el cliente y el servidor son procesos que ejecutan siempre en computadores diferentes.
- No obstante la red puede ser simulada y ambos procesos ejecutar en el mismo computador como si la red estuviese en medio.
- Los computadores pueden ser de diferente hardware (Intel uno y ARM el otro), y también pueden tener sistemas operativos diferentes (MAC uno y LINUX el otro).
- Sin embargo esto detalles quedan completamente en segundo plano.
- Los sistemas Operativos de los computadores ni siquiera necesitan ser POSIX ya que la interfaz de sockets es de mas alto nivel aun
- Sin embargo en POSIX el socket es tratado como un archivo, de modo que solo es necesario que si ambos sistemas son POSIX compatibles, no solo nos sirven los mismos fuentes sino que además podemos manejarlo muy fácilmente.

Caso general: Intercomunicación por red

- En este caso el cliente y el servidor son procesos que ejecutan siempre en computadores diferentes.
- No obstante la red puede ser simulada y ambos procesos ejecutar en el mismo computador como si la red estuviese en medio.
- Los computadores pueden ser de diferente hardware (Intel uno y ARM el otro), y también pueden tener sistemas operativos diferentes (MAC uno y LINUX el otro).
- Sin embargo esto detalles quedan completamente en segundo plano.
- Los sistemas Operativos de los computadores ni siquiera necesitan ser POSIX ya que la interfaz de sockets es de mas alto nivel aun
- Sin embargo en POSIX el socket es tratado como un archivo, de modo que solo es necesario que si ambos sistemas son POSIX compatibles, no solo nos sirven los mismos fuentes sino que además podemos manejarlo muy fácilmente.

Caso general: Intercomunicación por red

- En este caso el cliente y el servidor son procesos que ejecutan siempre en computadores diferentes.
- No obstante la red puede ser simulada y ambos procesos ejecutar en el mismo computador como si la red estuviese en medio.
- Los computadores pueden ser de diferente hardware (Intel uno y ARM el otro), y también pueden tener sistemas operativos diferentes (MAC uno y LINUX el otro).
- Sin embargo esto detalles quedan completamente en segundo plano.
- Los sistemas Operativos de los computadores ni siquiera necesitan ser POSIX ya que la interfaz de sockets es de mas alto nivel aun
- Sin embargo en POSIX el socket es tratado como un archivo, de modo que solo es necesario que si ambos sistemas son POSIX compatibles, no solo nos sirven los mismos fuentes sino que además podemos manejarlo muy fácilmente.

Caso general: Intercomunicación por red

- En este caso el cliente y el servidor son procesos que ejecutan siempre en computadores diferentes.
- No obstante la red puede ser simulada y ambos procesos ejecutar en el mismo computador como si la red estuviese en medio.
- Los computadores pueden ser de diferente hardware (Intel uno y ARM el otro), y también pueden tener sistemas operativos diferentes (MAC uno y LINUX el otro).
- Sin embargo esto detalles quedan completamente en segundo plano.
- Los sistemas Operativos de los computadores ni siquiera necesitan ser POSIX ya que la interfaz de sockets es de mas alto nivel aun
- Sin embargo en POSIX el socket es tratado como un archivo, de modo que solo es necesario que si ambos sistemas son POSIX compatibles, no solo nos sirven los mismos fuentes sino que además podemos manejarlo muy fácilmente.

1 Introducción

2 Conceptos de Networking

3 Interfaz de sockets

- ¿socket? ¿Que es?
- Internet sockets
- Modelo Cliente Servidor
- API de nuestra librería de alto nivel
- API de sockets (POSIX)
- Hands On

Solo tres funciones

- Además para simplificar mas las cosas podemos desarrollar un API de mas alto nivel que contienen las llamadas POSIX.
- En el caso de nuestro curso las funciones del API están definidas en el header sock-lib.h.
- Lado Server

```
1 int Open_conection (struct sockaddr_in *);  
2 int Aceptar_pedidos (int);
```

- Lado cliente

```
1     int      conectar (int, char **);
```

Open_connection

- Es la primer función que ejecuta el server.
- Recibe el puntero a una estructura en la que almacenará la dirección IP y el port de la conexión.
 - Crea el socket
 - Lo inicializa y lo enlaza con el sistema operativo para que pueda ser usado
 - Establece algunas configuraciones adicionales
- Devuelve un file descriptor para el socket creado de modo que el resto de las operaciones se refieran a este como si fuese un nodo más del file system.

Aceptar_pedidos

- Luego de Open_connection, solo queda esperar que se reciba un pedido desde un cliente para resolver y responder.
 - Espera que llegue algo por el socket
 - Al recibir algo crea un duplicado del socket para que use pueda transaccionar con el cliente por este socket duplicado, y devuelve ese socket.
 - Por el nuevo socket no se pueden aceptar conexiones. Es solo para transaccionar.
 - También se puede responder por el socket original. Pero veremos mas adelante que lo ideal es desdobljar el procesamiento para volver a esperar mientras se transacciona con el cliente.

conectar

- El cliente solo ejecuta esta función. Cuando devuelve el control comienza a transaccionar con el server.
 - Conecta con el server. La dirección va en el primer elemento del puntero a char. El port va en el segundo
 - Previamente crea un socket.
 - Por el nuevo socket no se pueden aceptar conexiones. Es solo para transaccionar.
 - También se puede responder por el socket original. Pero veremos mas adelante que lo ideal es desdoblar el procesamiento para volver a esperar mientras se transacciona con el cliente.

1 Introducción

2 Conceptos de Networking

3 Interfaz de sockets

- ¿socket? ¿Que es?
- Internet sockets
- Modelo Cliente Servidor
- API de nuestra librería de alto nivel
- **API de sockets (POSIX)**
- Hands On

Estructuras de datos

- Una de las estructuras core es la que define la dirección de internet.

```
1 struct sockaddr {  
2     unsigned short sa_family;  
3     char    sa_data[14];  
4 };
```

- **sa_family**: Es la familia de direcciones. Para el uso que le pensamos dar es **AF_INET**.
- **sa_data**: Contiene en general la dirección de internet y el puerto.
- La razón de estos 14 bytes es que la estructura sirva para cualquier implementación no importa el tamaño de dato que utilice para guardar la dirección y el port.
- Es tedioso lidiar con **sa_data** ingresando allí el port y la dirección "a mano".

Estructuras de datos

- Una de las estructuras core es la que define la dirección de internet.

```
1 struct sockaddr {  
2     unsigned short sa_family;  
3     char    sa_data[14];  
4 };
```

- sa_family:** Es la familia de direcciones. Para el uso que le pensamos dar es **AF_INET**.
- sa_data:** Contiene en general la dirección de internet y el puerto.
- La razón de estos 14 bytes es que la estructura sirva para cualquier implementación no importa el tamaño de dato que utilice para guardar la dirección y el port.
- Es tedioso lidiar con **sa_data** ingresando allí el port y la dirección "a mano".

Estructuras de datos

- Una de las estructuras core es la que define la dirección de internet.

```
1 struct sockaddr {  
2     unsigned short sa_family;  
3     char    sa_data[14];  
4 };
```

- **sa_family**: Es la familia de direcciones. Para el uso que le pensamos dar es **AF_INET**.
- **sa_data**: Contiene en general la dirección de internet y el puerto.
- La razón de estos 14 bytes es que la estructura sirva para cualquier implementación no importa el tamaño de dato que utilice para guardar la dirección y el port.
- Es tedioso lidiar con **sa_data** ingresando allí el port y la dirección “a mano”.

Estructuras de datos

- Una de las estructuras core es la que define la dirección de internet.

```
1 struct sockaddr {  
2     unsigned short sa_family;  
3     char    sa_data[14];  
4 };
```

- **sa_family**: Es la familia de direcciones. Para el uso que le pensamos dar es **AF_INET**.
- **sa_data**: Contiene en general la dirección de internet y el puerto.
- La razón de estos 14 bytes es que la estructura sirva para cualquier implementación no importa el tamaño de dato que utilice para guardar la dirección y el port.
- Es tedioso lidiar con **sa_data** ingresando allí el port y la dirección “a mano”.

Estructuras de datos

- Una de las estructuras core es la que define la dirección de internet.

```
1 struct sockaddr {  
2     unsigned short sa_family;  
3     char    sa_data[14];  
4 };
```

- sa_family**: Es la familia de direcciones. Para el uso que le pensamos dar es **AF_INET**.
- sa_data**: Contiene en general la dirección de internet y el puerto.
- La razón de estos 14 bytes es que la estructura sirva para cualquier implementación no importa el tamaño de dato que utilice para guardar la dirección y el port.
- Es tedioso lidiar con **sa_data** ingresando allí el port y la dirección “a mano”.

Estructuras de datos

- Por eso para el caso de **AF_INET** se utiliza una variante de esta estructura.

```
1 struct sockaddr_in {  
2     short int           sin_family;  
3     unsigned short int sin_port;  
4     struct in_addr       sin_addr;  
5     unsigned char        sin_zero[8];  
6 };
```

- Esta estructura simplifica el acceso a los diferentes elementos que deben completarse para definir un socket y su dirección y puerto.

- La dirección se define mediante una estructura:

```
1 struct in_addr {  
2     uint32_t s_addr;  
3 };
```

- ¿Porque?

Estructuras de datos

- Por eso para el caso de **AF_INET** se utiliza una variante de esta estructura.

```
1 struct sockaddr_in {  
2     short int           sin_family;  
3     unsigned short int sin_port;  
4     struct in_addr       sin_addr;  
5     unsigned char        sin_zero[8];  
6 };
```

- Esta estructura simplifica el acceso a los diferentes elementos que deben completarse para definir un socket y su dirección y puerto.

- La dirección se define mediante una estructura:

```
1 struct in_addr {  
2     uint32_t s_addr;  
3 };
```

- ¿Porque?

Estructuras de datos

- Por eso para el caso de **AF_INET** se utiliza una variante de esta estructura.

```
1 struct sockaddr_in {  
2     short int           sin_family;  
3     unsigned short int sin_port;  
4     struct in_addr       sin_addr;  
5     unsigned char        sin_zero[8];  
6 };
```

- Esta estructura simplifica el acceso a los diferentes elementos que deben completarse para definir un socket y su dirección y puerto.
- La dirección se define mediante una estructura:

```
1 struct in_addr {  
2     uint32_t s_addr;  
3 };
```

- ¿Porque?

Estructuras de datos

- Por eso para el caso de **AF_INET** se utiliza una variante de esta estructura.

```
1 struct sockaddr_in {  
2     short int           sin_family;  
3     unsigned short int sin_port;  
4     struct in_addr       sin_addr;  
5     unsigned char        sin_zero[8];  
6 };
```

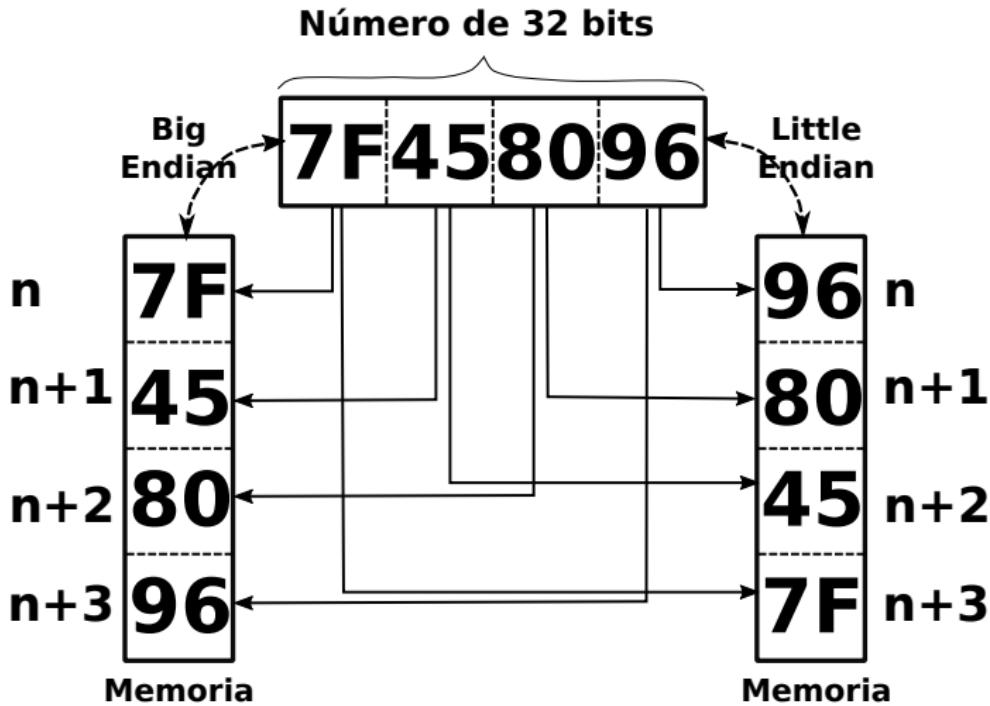
- Esta estructura simplifica el acceso a los diferentes elementos que deben completarse para definir un socket y su dirección y puerto.
- La dirección se define mediante una estructura:

```
1 struct in_addr {  
2     uint32_t s_addr;  
3 };
```

- ¿Porque?

Byte Order

- Cuando representamos un valor de varios bytes el orden en el que se almacena no es trivial.



Byte Order

- En su momento los diseñadores de TCP/IP adoptaron el formato Big Endian para ordenar los valores de mas de un byte en un paquete definido por el protocolo para su transmisión.
- Centrados en el problema de transmisión de datos se denominó al formato Big Endian, Network Order Byte.
- Este formato es estándar en el mundo del networking
- No así en el universo de los computadores, en el que está bastante dividido.
- De hecho los procesadores IA-32 e Intel®64, almacenan los datos en memoria como Little Endian
- Por ello es que se habla de Network Order Byte y Host Order Byte. en el primero no hay dudas pero el segundo no es estándar.
- Por tal motivo existe la función `htons()`, host to network short para convertir un número en formato Host Order Byte a Network Order Byte.

Byte Order

- En su momento los diseñadores de TCP/IP adoptaron el formato Big Endian para ordenar los valores de mas de un byte en un paquete definido por el protocolo para su transmisión.
- Centrados en el problema de transmisión de datos se denominó al formato Big Endian, Network Order Byte.
- Este formato es estándar en el mundo del networking
- No así en el universo de los computadores, en el que está bastante dividido.
- De hecho los procesadores IA-32 e Intel®64, almacenan los datos en memoria como Little Endian
- Por ello es que se habla de Network Order Byte y Host Order Byte. en el primero no hay dudas pero el segundo no es estándar.
- Por tal motivo existe la función `htons()`, host to network short para convertir un número en formato Host Order Byte a Network Order Byte.

Byte Order

- En su momento los diseñadores de TCP/IP adoptaron el formato Big Endian para ordenar los valores de mas de un byte en un paquete definido por el protocolo para su transmisión.
- Centrados en el problema de transmisión de datos se denominó al formato Big Endian, Network Order Byte.
- Este formato es estándar en el mundo del networking
- No así en el universo de los computadores, en el que está bastante dividido.
- De hecho los procesadores IA-32 e Intel®64, almacenan los datos en memoria como Little Endian
- Por ello es que se habla de Network Order Byte y Host Order Byte. en el primero no hay dudas pero el segundo no es estándar.
- Por tal motivo existe la función `htons()`, host to network short para convertir un número en formato Host Order Byte a Network Order Byte.

Byte Order

- En su momento los diseñadores de TCP/IP adoptaron el formato Big Endian para ordenar los valores de mas de un byte en un paquete definido por el protocolo para su transmisión.
- Centrados en el problema de transmisión de datos se denominó al formato Big Endian, Network Order Byte.
- Este formato es estándar en el mundo del networking
- No así en el universo de los computadores, en el que está bastante dividido.
- De hecho los procesadores IA-32 e Intel®64, almacenan los datos en memoria como Little Endian
- Por ello es que se habla de Network Order Byte y Host Order Byte. en el primero no hay dudas pero el segundo no es estándar.
- Por tal motivo existe la función `htons()`, host to network short para convertir un número en formato Host Order Byte a Network Order Byte.

Byte Order

- En su momento los diseñadores de TCP/IP adoptaron el formato Big Endian para ordenar los valores de mas de un byte en un paquete definido por el protocolo para su transmisión.
- Centrados en el problema de transmisión de datos se denominó al formato Big Endian, Network Order Byte.
- Este formato es estándar en el mundo del networking
- No así en el universo de los computadores, en el que está bastante dividido.
- De hecho los procesadores IA-32 e Intel®64, almacenan los datos en memoria como Little Endian
- Por ello es que se habla de Network Order Byte y Host Order Byte. en el primero no hay dudas pero el segundo no es estándar.
- Por tal motivo existe la función `htons()`, host to network short para convertir un número en formato Host Order Byte a Network Order Byte.

Byte Order

- En su momento los diseñadores de TCP/IP adoptaron el formato Big Endian para ordenar los valores de mas de un byte en un paquete definido por el protocolo para su transmisión.
- Centrados en el problema de transmisión de datos se denominó al formato Big Endian, Network Order Byte.
- Este formato es estándar en el mundo del networking
- No así en el universo de los computadores, en el que está bastante dividido.
- De hecho los procesadores IA-32 e Intel®64, almacenan los datos en memoria como Little Endian
- Por ello es que se habla de Network Order Byte y Host Order Byte. en el primero no hay dudas pero el segundo no es estándar.
- Por tal motivo existe la función `htons()`, host to network short para convertir un número en formato Host Order Byte a Network Order Byte.

Byte Order

- En su momento los diseñadores de TCP/IP adoptaron el formato Big Endian para ordenar los valores de mas de un byte en un paquete definido por el protocolo para su transmisión.
- Centrados en el problema de transmisión de datos se denominó al formato Big Endian, Network Order Byte.
- Este formato es estándar en el mundo del networking
- No así en el universo de los computadores, en el que está bastante dividido.
- De hecho los procesadores IA-32 e Intel®64, almacenan los datos en memoria como Little Endian
- Por ello es que se habla de Network Order Byte y Host Order Byte. en el primero no hay dudas pero el segundo no es estándar.
- Por tal motivo existe la función **hton_s ()**, host to network short para convertir un número en formato Host Order Byte a Network Order Byte.

Syscalls para convertir formatos

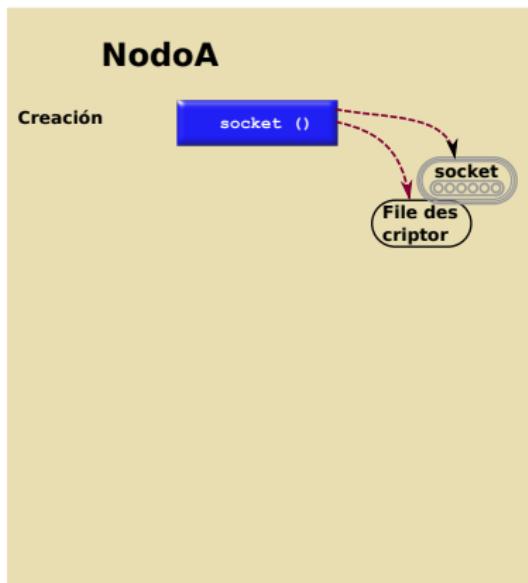
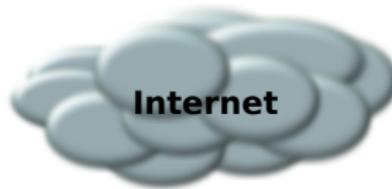
Network a Host

- `uint16_t htons(uint16_t hostshort)`: host to network short
- `uint32_t htonl(uint32_t hostlong)`: host to network long
- `uint16_t ntohs(uint16_t netshort)`: network to host short
- `uint32_t ntohl(uint32_t netlong)`: network to host long

Para formatear direcciones

- `in_addr_t inet_addr(const char *cp)`: Convierte una dirección en notación punto a un entero de 32 bits en formato Host.
- `int inet_aton(const char *cp, struct in_addr *inp)`: Convierte una cadena ascii con notación punto y la guarda en `inp`,
- `char *inet_ntoa(struct in_addr in)`: Obtienen la notación punto a partir de un entero en formato network.

Syscalls principales



Syscalls principales

socket ()

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

Crea un objeto **socket** en el sistema y obtiene su file descriptor. Este se comporta como un conector a un canal de comunicaciones bidireccional. Tiene tres argumentos

- ① **domain**. Especifica un dominio de comunicaciones (familia de protocolos definida en **<sys/socket.h>**): **PF_INET** (o **AF_INET** que es equivalente) es el que vamos a trabajar en nuestro curso (IP v4).

Syscalls principales

socket ()

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

Crea un objeto **socket** en el sistema y obtiene su file descriptor. Este se comporta como un conector a un canal de comunicaciones bidireccional. Tiene tres argumentos

- ② **type**. Tipo de socket: **SOCK_STREAM**, **SOCK_DGRAM**, **SOCK_RAW**, **SOCK_SEQPACKET**, entre otros. Nos vamos a concentrar en los dos primeros.

Syscalls principales

socket ()

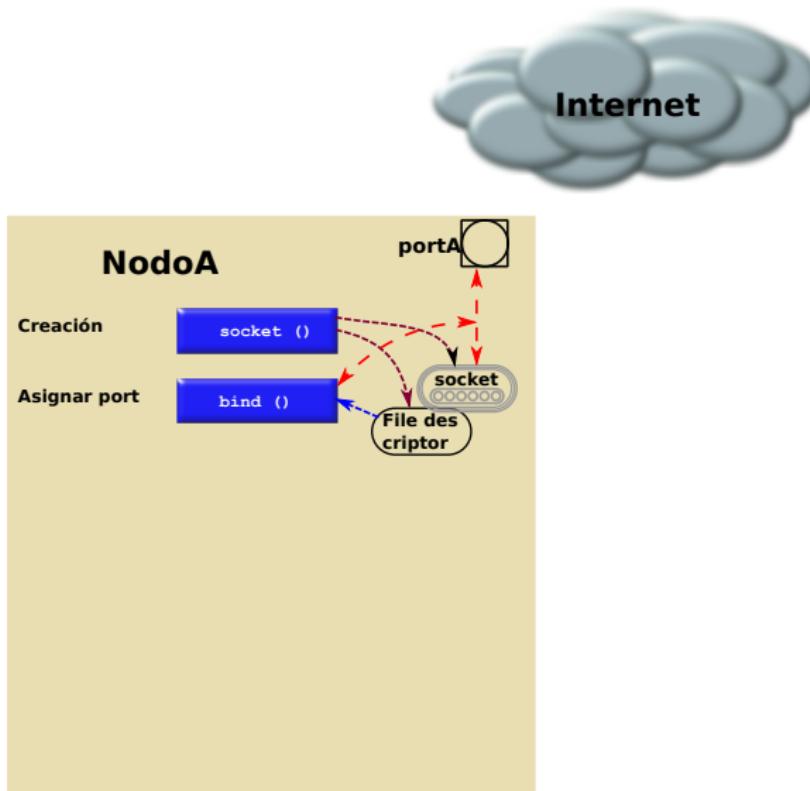
```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

Crea un objeto **socket** en el sistema y obtiene su file descriptor. Este se comporta como un conector a un canal de comunicaciones bidireccional. Tiene tres argumentos

- ③ **protocol**. Una vez definid dominio y tipo, puede existir en ocasiones varios protocolos. Normalmente no es así con lo cual es muy común que este argumento sea 0.

Syscalls principales



Syscalls principales

bind ()

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int s, struct sockaddr *my_addr, int addrlen);
```

Le asigna al objeto **socket** que recibe en el primer argumento, un par dirección ip : puerto.

- ① **s**. File descriptor del creado por la sys call **socket ()**.
- ② **my_addr**. Es el puntero a una estructura **sockaddr**, en donde se le escriben el par dirección ip: puerto. Normalmente, nos manejamos con la estructura **sockaddr_in**, en la que están definidos los miembros **sin_addr** y **sin_port**. En tal caso no hay que omitir el cast.

Syscalls principales

bind ()

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int s, struct sockaddr *my_addr, int addrlen);
```

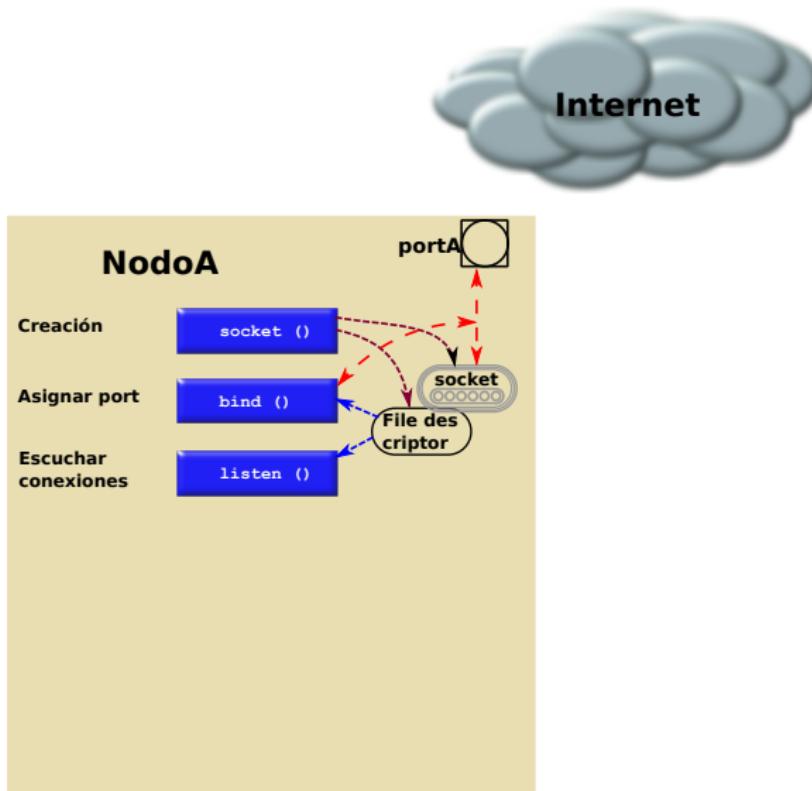
Le asigna al objeto **socket** que recibe en el primer argumento, un par dirección ip : puerto.

- ③ **addrlen**. Es la longitud en bytes de **my_addr**. Conviene utilizar el operador **sizeof()** para asegurar portabilidad interplataformas.

Juntando parte de lo visto

```
1 #include <string.h>
2 #include <sys/types.h>
3 #include <sys/socket.h>
4 #include <netinet/in.h>
5 #define PORT 9876
6 int main(void)
7 {
8     int sfd;
9     struct sockaddr_in mi_addr;
10    if(((sfd = socket(PF_INET, SOCK_STREAM, 0))< 0)
11        exit_on_error();
12    mi_addr.sin_family = AF_INET;
13    mi_addr.sin_port = htons(MYPORT); //host byte order
14    mi_addr.sin_addr.s_addr = INADDR_ANY //ip local
15    bzero(&(my_addr->sin_zero), 8); /* rellena con ceros*/
16    if (bind(sfd, (struct sockaddr *)&mi_addr, sizeof(
17        mi_addr))<0)
18        exit_on_error();
```

Syscalls principales



Syscalls principales

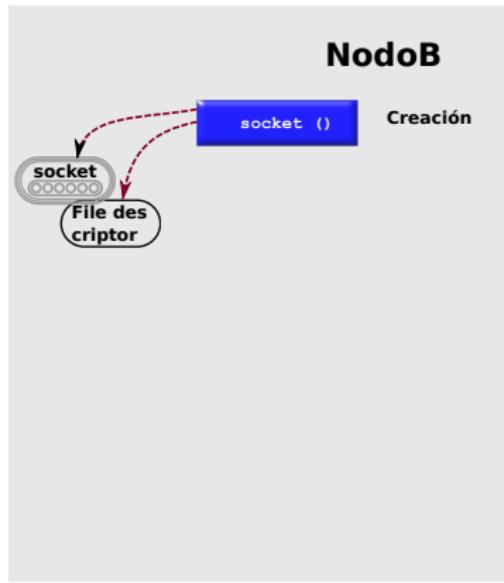
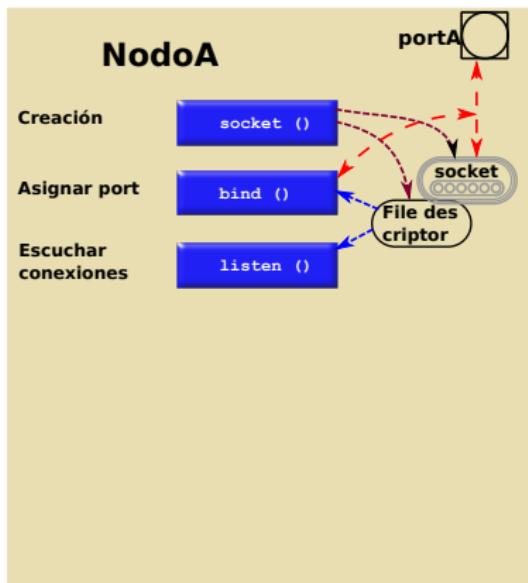
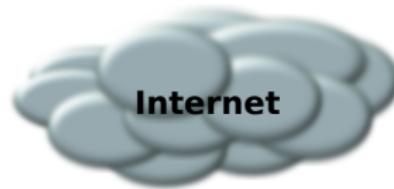
listen ()

```
#include <sys/socket.h>

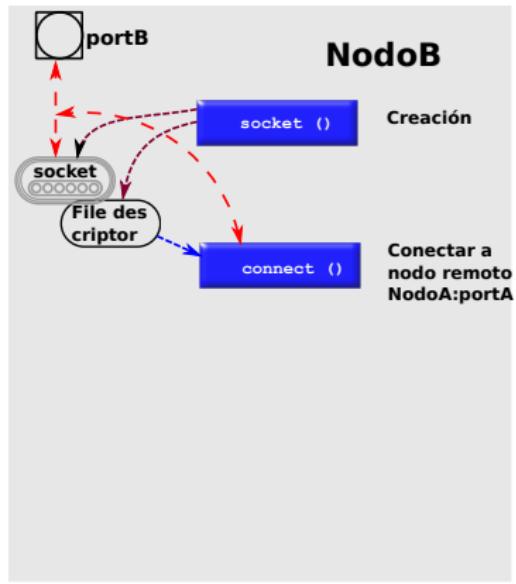
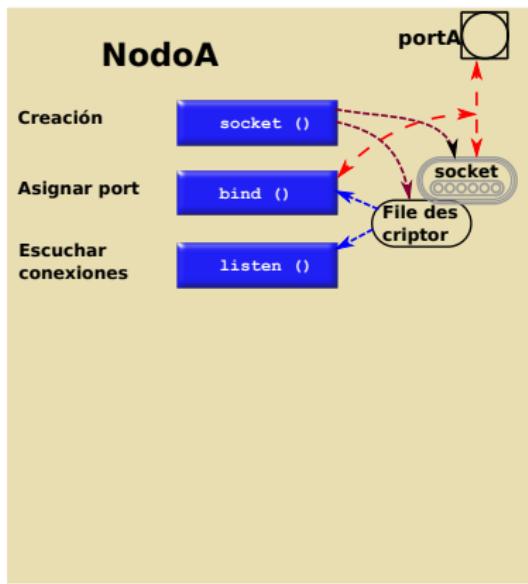
int listen(int s, int backlog);
```

- ① **s**. File descriptor del creado por la sys call **socket ()**.
 - ② **backlog**. Es la cantidad de pedidos de conexión que el proceso almacenará mientras se responde al pedido de conexión en curso de ser aceptado.
-
- **listen ()** trabaja en conjunto con la llamada **accept ()**.
 - Si bien no bloquea al proceso (de eso se encarga **accept ()**), tiene dentro de la estructura de datos del socket una función callback llamada **inet_listen ()**, que se ejecuta en medio del three way handshake junto con la callback **inet_accept ()** para completar la resolución de la conexión.

Syscalls principales



Syscalls principales



Syscalls principales

listen ()

```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int s, struct sockaddr *serv_addr, int
            addrlen);
```

Pone al proceso a la escucha de conexiones provenientes del socket pasado como argumento. El estado es **LISTENING**. Aplica solo a sockets tipo **SOCK_STREAM** o **SOCK_SEQPACKET**.

- ① **s**. File descriptor del creado por la sys call **socket ()**.
- ② **serv_addr**. Es el puntero a una estructura **sockaddr**, en donde se le escriben el par dirección ip: puerto del server al que se desea conectar. Normalmente, nos manejamos con la estructura **sockaddr_in**, en la que están definidos los miembros **sin_addr** y **sin_port**. En tal caso no hay que omitir el cast.

Syscalls principales

listen ()

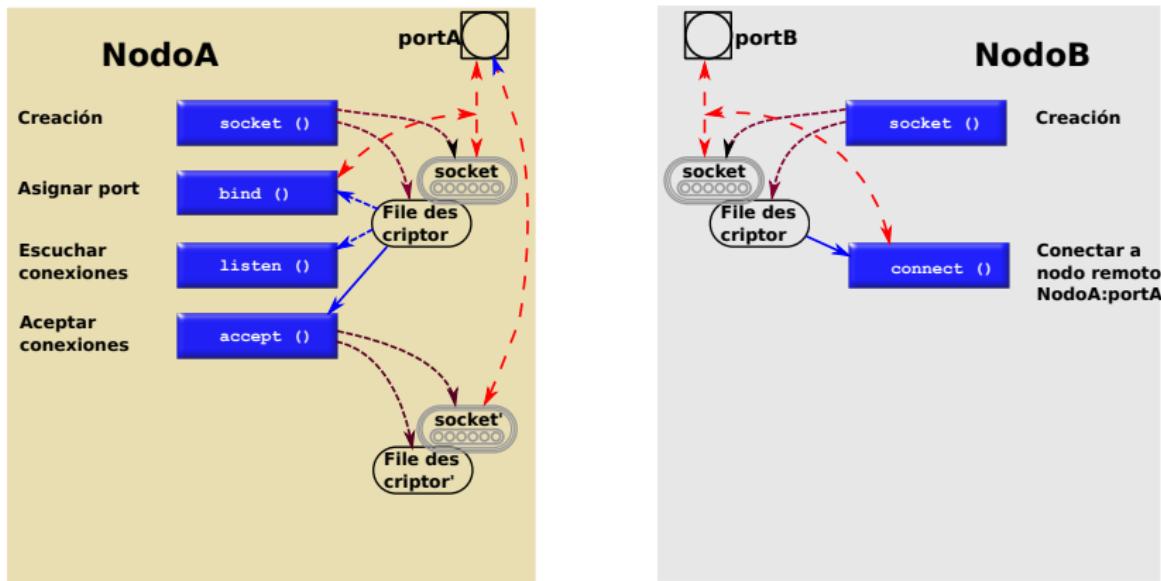
```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int s, struct sockaddr *serv_addr, int
            addrlen);
```

Pone al proceso a la escucha de conexiones provenientes del socket pasado como argumento. El estado es **LISTENING**. Aplica solo a sockets tipo **SOCK_STREAM** o **SOCK_SEQPACKET**.

- ③ **addrlen**. Es la longitud en bytes de **my_addr**. Conviene utilizar el operador **sizeof()** para asegurar portabilidad interplataformas.

Syscalls principales



Syscalls principales

accept ()

```
#include <sys/types.h>
#include <sys/socket.h>

int accept(int s, struct sockaddr *addr, socklen_t *
           addrlen);
```

Extrae el primer pedido de conexión de la cola de conexiones pendientes del socket **s**, le asocia un nuevo socket que es un duplicado de **s** y le reserva un nuevo file descriptor. Éste es el valor devuelto por la llamada. Aplica a sockets del tipo SOCK_STREAM, SOCK_SEQPACKET y SOCK_RDM.

- ① **s**. File descriptor del creado por la sys call **socket ()**.

Syscalls principales

accept ()

```
#include <sys/types.h>
#include <sys/socket.h>

int accept(int s, struct sockaddr *addr, socklen_t *
           addrlen);
```

Extrae el primer pedido de conexión de la cola de conexiones pendientes del socket **s**, le asocia un nuevo socket que es un duplicado de **s** y le reserva un nuevo file descriptor. Éste es el valor devuelto por la llamada. Aplica a sockets del tipo SOCK_STREAM, SOCK_SEQPACKET y SOCK_RDM.

- ② **addr**. Es el puntero a una estructura **sockaddr**, en donde se le escriben el par dirección ip: puerto del nodo que requirió la conexión. Normalmente, nos manejamos con la estructura **sockaddr_in**, en la que están definidos los miembros **sin_addr** y **sin_port**. En tal caso no hay que omitir el cast.

Syscalls principales

accept ()

```
#include <sys/types.h>
#include <sys/socket.h>

int accept(int s, struct sockaddr *addr, socklen_t *
           addrlen);
```

Extrae el primer pedido de conexión de la cola de conexiones pendientes del socket **s**, le asocia un nuevo socket que es un duplicado de **s** y le reserva un nuevo file descriptor. Éste es el valor devuelto por la llamada. Éste es el valor devuelto por la llamada. Aplica a sockets del tipo SOCK_STREAM, SOCK_SEQPACKET y SOCK_RDM.

- ③ **addrlen**. Es la longitud en bytes de **addr**. Conviene utilizar el operador **sizeof()** para asegurar portabilidad interplataformas.

Syscalls principales

accept (). ¿Porque un duplicado del socket original?.

- En realidad no es un duplicado exacto.
- La diferencia sustancial es que este segundo socket difiere del original en que no está en estado **LISTENING**. Por lo tanto *no es apto para escuchar nuevas conexiones*. Su uso es para intercambiar datos con el nodo extremo.
- El socket original del proceso, que fue creado por **socket ()**, asignado al port por **bind ()**, y utilizado por **listen ()** para escuchar solicitudes de conexión, no sufre alteración alguna.
- De este modo, el socket original puede volver a escuchar conexiones mientras se utiliza el “duplicado” para intercambiar conexiones con el extremo remoto cuya solicitud de conexión ha sido **accept ()**ada.

Syscalls principales

accept (). ¿Porque un duplicado del socket original?.

- En realidad no es un duplicado exacto.
- La diferencia sustancial es que este segundo socket difiere del original en que no está en estado **LISTENING**. Por lo tanto *no es apto para escuchar nuevas conexiones*. Su uso es para intercambiar datos con el nodo extremo.
- El socket original del proceso, que fue creado por **socket ()**, asignado al port por **bind ()**, y utilizado por **listen ()** para escuchar solicitudes de conexión, no sufre alteración alguna.
- De este modo, el socket original puede volver a escuchar conexiones mientras se utiliza el “duplicado” para intercambiar conexiones con el extremo remoto cuya solicitud de conexión ha sido **accept ()**ada.

Syscalls principales

accept (). ¿Porque un duplicado del socket original?.

- En realidad no es un duplicado exacto.
- La diferencia sustancial es que este segundo socket difiere del original en que no está en estado **LISTENING**. Por lo tanto *no es apto para escuchar nuevas conexiones*. Su uso es para intercambiar datos con el nodo extremo.
- El socket original del proceso, que fue creado por `socket ()`, asignado al port por `bind ()`, y utilizado por `listen ()` para escuchar solicitudes de conexión, no sufre alteración alguna.
- De este modo, el socket original puede volver a escuchar conexiones mientras se utiliza el “duplicado” para intercambiar conexiones con el extremo remoto cuya solicitud de conexión ha sido `accept ()`ada.

Syscalls principales

accept (). ¿Porque un duplicado del socket original?.

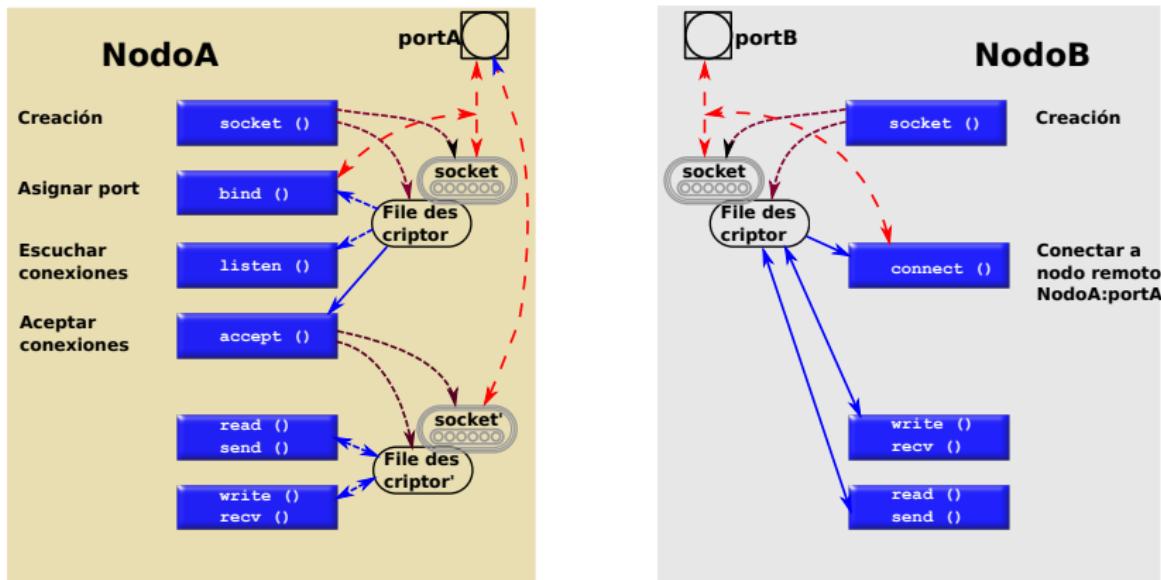
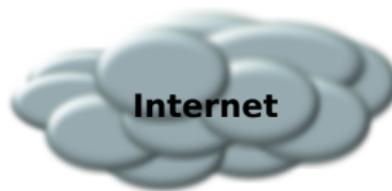
- En realidad no es un duplicado exacto.
- La diferencia sustancial es que este segundo socket difiere del original en que no está en estado **LISTENING**. Por lo tanto *no es apto para escuchar nuevas conexiones*. Su uso es para intercambiar datos con el nodo extremo.
- El socket original del proceso, que fue creado por **socket ()**, asignado al port por **bind()**, y utilizado por **listen()** para escuchar solicitudes de conexión, no sufre alteración alguna.
- De este modo, el socket original puede volver a escuchar conexiones mientras se utiliza el “duplicado” para intercambiar conexiones con el extremo remoto cuya solicitud de conexión ha sido **accept ()** ada.

Syscalls principales

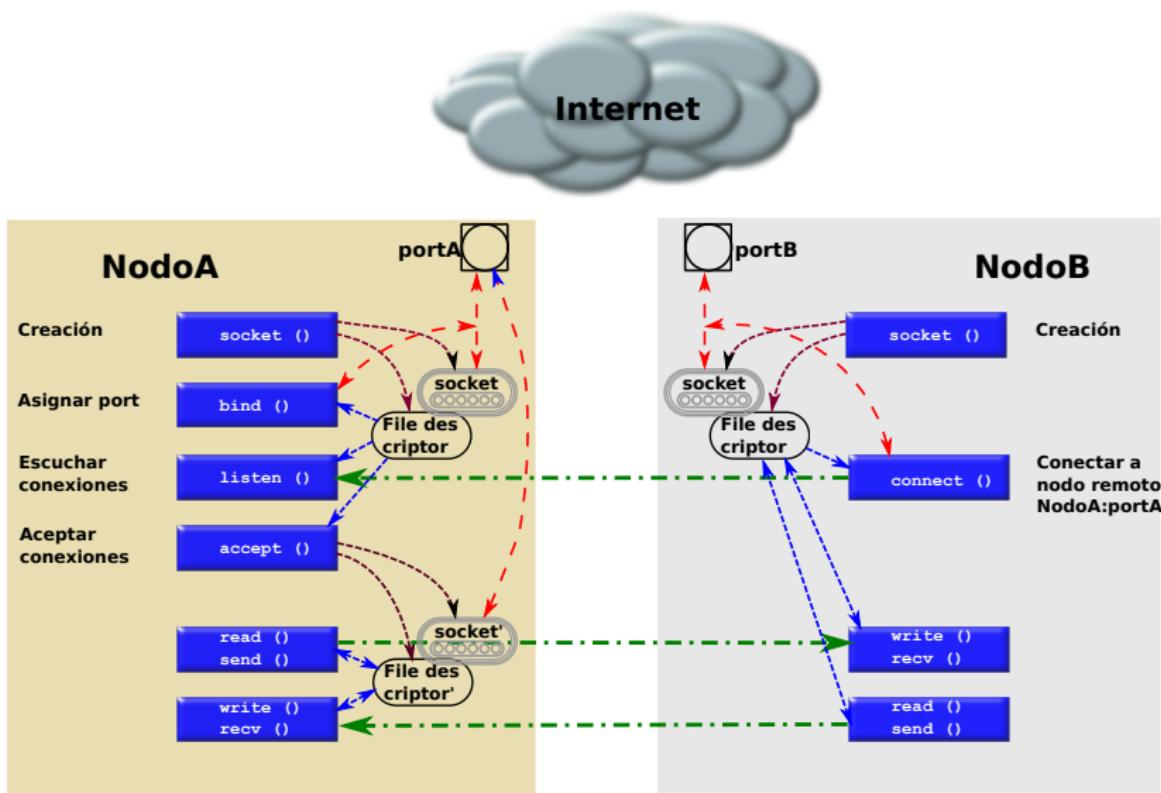
accept (). ¿Porque un duplicado del socket original?.

- En realidad no es un duplicado exacto.
- La diferencia sustancial es que este segundo socket difiere del original en que no está en estado **LISTENING**. Por lo tanto *no es apto para escuchar nuevas conexiones*. Su uso es para intercambiar datos con el nodo extremo.
- El socket original del proceso, que fue creado por **socket ()**, asignado al port por **bind()**, y utilizado por **listen()** para escuchar solicitudes de conexión, no sufre alteración alguna.
- De este modo, el socket original puede volver a escuchar conexiones mientras se utiliza el “duplicado” para intercambiar conexiones con el extremo remoto cuya solicitud de conexión ha sido **accept ()**ada.

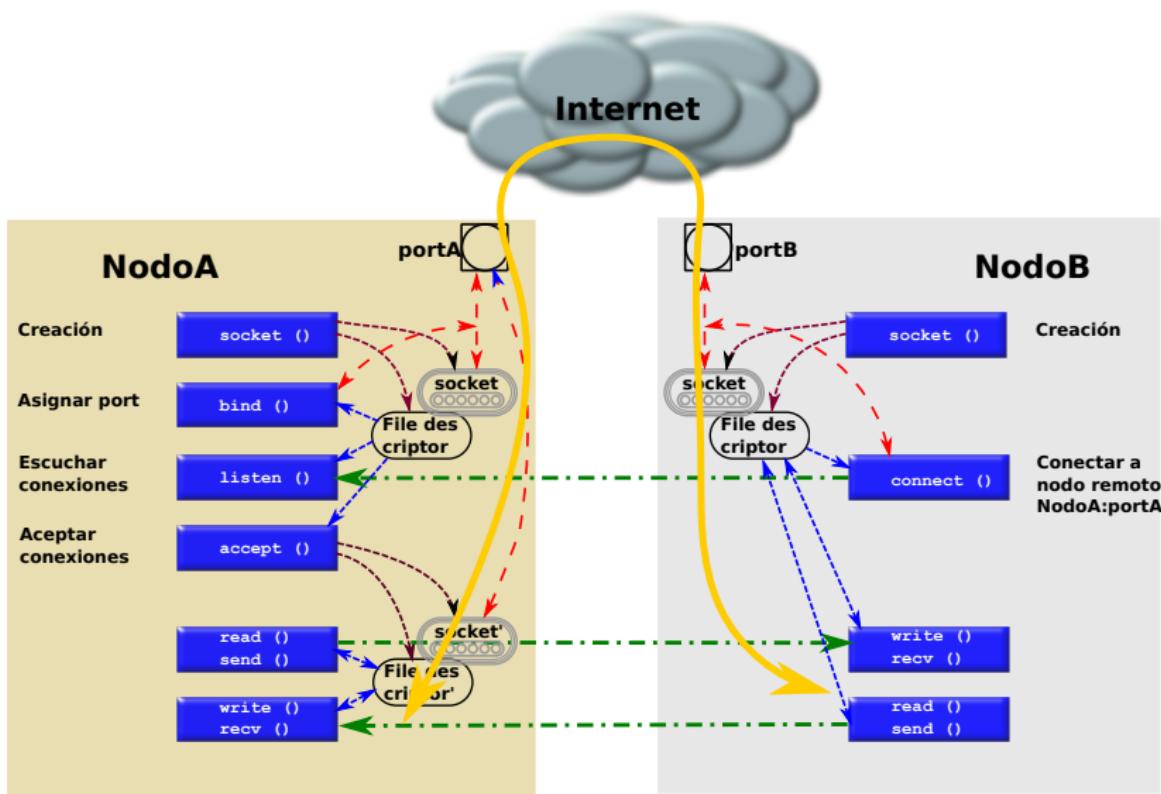
Syscalls principales



Syscalls principales



Syscalls principales



1 Introducción

2 Conceptos de Networking

3 Interfaz de sockets

- ¿socket? ¿Que es?
- Internet sockets
- Modelo Cliente Servidor
- API de nuestra librería de alto nivel
- API de sockets (POSIX)
- Hands On

Hands On!