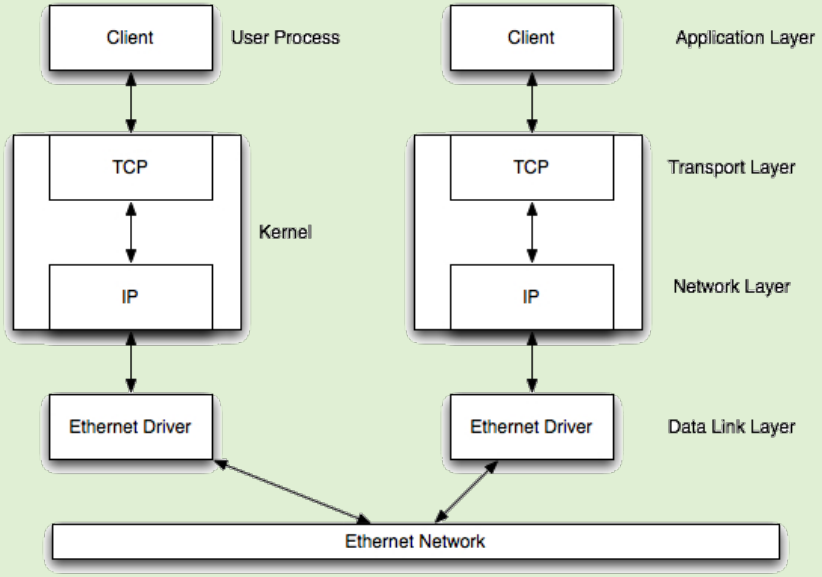
Taller de Programación en Redes
Stack TCP/IP - Sockets



Lic. en Sistemas de Información - Universidad Nacional de Luján
Dr. Gabriel Tolosa – tolosoft@unlu.edu.ar
Lic. Marcelo Fernández – fernandezm@unlu.edu.ar
Clase 4 - Febrero 2018

Sockets Asincrónicos – Modelo de trabajo

1. Creo el socket, definiéndole el timeout en 0.

2. Se entra en un loop.

3. En el inicio del loop se espera por eventos de I/O, mediante una llamada bloqueante.

4. En el caso de un evento (o más de uno), la llamada bloqueante sale, devolviendo una lista de sockets/eventos asociados.

5. Se recorre la lista de eventos de entrada/salida y por cada evento se llama a la rutina correspondiente para atenderlo.

6. Vuelta a 1.

Sockets Asincrónicos – `select()` system call

`select.`**`select`**(*rlist*, *wlist*, *xlist*[, *timeout*])

This is a straightforward interface to the Unix **`select()`** system call. The first three arguments are sequences of 'waitable objects': either integers representing file descriptors or objects with a parameterless method named `fileno()` returning such an integer:

- *rlist*: wait until ready for reading
- *wlist*: wait until ready for writing
- *xlist*: wait for an "exceptional condition" (see the manual page for what your system considers such a condition)

Empty sequences are allowed, but acceptance of three empty sequences is platform-dependent. (It is known to work on Unix but not on Windows.) The optional *timeout* argument specifies a time-out as a floating point number in seconds. When the *timeout* argument is omitted the function blocks until at least one file descriptor is ready. A time-out value of zero specifies a poll and never blocks.

The return value is a triple of lists of objects that are ready: subsets of the first three arguments. When the time-out is reached without a file descriptor becoming ready, three empty lists are returned.

Among the acceptable object types in the sequences are Python file objects (e.g. `sys.stdin`, or objects returned by `open()` or `os.popen()`), socket objects returned by `socket.socket()`. You may also define a *wrapper* class yourself, as long as it has an appropriate `fileno()` method (that really returns a file descriptor, not just a random integer).

Sockets Asincrónicos – `select()` system call

Algunas observaciones:

- Select hoy en día no es el método más eficiente, ni el único:

  1) Poll, Epoll en Unix/Linux

  2) Kqueue, Kevent en *BSD

  3) IOCP en Windows [1], AIX y Solaris 10+

  4) Abstracciones (no son syscalls): Libevent, Libev, Libuv

…Pero select sigue siendo uno de los más antiguos y portable entre plataformas.

Breve comparativa entre select, poll y epoll (y por qué los dos primeros son lentos):
https://jvns.ca/blog/2017/06/03/async-io-on-linux--select--poll--and-epoll/

Sockets Asincrónicos – `select()` system call

Ejemplos de código

URL: https://pymotw.com/2/select/
Sources: https://github.com/douglas/pymotw-br/tree/master/PyMOTW/socket

Scapy

Tutorial Interactivo

http://scapy.readthedocs.io/en/latest/usage.html#interactive-tutorial

Sockets Asincrónicos – Ejercicios Propuestos

1) Codificar un server que maneje una BD centralizada[1] en SQLite y tenga 'n' clientes que se conecten y le manden queries SQL, devolviendo las respuestas correspondientes según la BD local [2][3].

2) Programar una especie de servidor de nombres pero orientado a la conexión. Es decir, el servidor lee el archivo /etc/hosts y tiene una tabla de nombres. Luego, los clientes le "piden" resolución de nombres. Si el servidor no lo tiene, hace una consulta DNS verdadera, actualiza su caché y responde. Se podría poner una restricción que el servidor tenga la posibilidad de atender unos pocos clientes en simultáneo (por ejemplo, 3) y cada cliente puede hacer unas pocas consultas en ráfaga (una especie de “cuota” parametrizable). Luego de cumplida la cuota el server le cierra la conexión y habilita al que está esperando (se puede relacionar con el concepto de *backlog* en TCP).

3) Escriba un programa que implemente un anillo lógico con dos *tokens* recorriéndolo, uno en cada sentido. Cada proceso en un host diferente tiene un ID (único) y cada uno puede transmitir cuando le llega el turno (secuencial). Cuando se envía un mensaje se debe incluir a qué ID va dirigido y siempre se reenvía en anillo – por ejemplo – hacia el nodo de la izquierda o la derecha. El mensaje debe circular hasta llegar nuevamente al nodo que lo originó quien dará lugar a que transmita el siguiente (similar al protocolo Token Ring). Cada nodo lee de un archivo la lista de mensajes que tiene que transmitir. Los programas finalizan su ejecución cuando el nodo con ID = 0 envía un comando de finalización. Usted debe definir el protocolo de aplicación, tanto la estructura de datos como el comportamiento ante cada situación.

[1] M3.0+ Earthquakes in the Contiguous U.S., 1995 through 2015, SQLite database.
http://2016.padjo.org/tutorials/sqlite-data-starterpacks/#more-info-m3-0-earthquakes-in-the-contiguous-u-s-1995-through-2015
[2] Paquete ‘sqlitebrowser’ en Debian/Ubuntu - http://sqlitebrowser.org/
[3] Módulo sqlite3 de Python - https://docs.python.org/2/library/sqlite3.html