UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA

DEPARTAMENTO DE ELECTRÓNICA

ELO322 – Redes de Computadores 1

**INTERFAZ CLIENTE/SERVIDOR PARA INTERCAMBIO DE INFORMACIÓN A DISTANCIA POR MEDIO DE PROTOCOLO TCP/IP**

REDES DE COMPUTADORES

Cristóbal Barrientos

Gonzalo Rojas

Claudio Sotomayor

Profesor: Agustín González

30 de julio de 2012

## Resumen

Hoy en día es necesario que las comunicaciones sean eficientes, simultáneas y en tiempo real, bajo esta premisa queremos controlar nuestros dispositivos a distancia, como sistemas de alarmas que nos protejan de asaltos, cámaras de seguridad para conocer qué está pasando en nuestro entorno cuando no estamos, etc. Desde cualquier parte, en nuestra oficina, al viajar, u otros lugares.

Nuestro trabajo aborda estos tópicos, explicando una solución relacionada con la programación en un lenguaje específico (el utilizado en este caso es python) de un servidor capaz de controlar distintos elementos dentro de una red al cual todos los usuarios estarán conectados, a través de una adecuada programación de Sockets y uso de “hebras” o hilos de ejecución.

## Objetivos

* Aplicar los conocimientos adquiridos en la asignatura Elo322 orientados hacia un proyecto, el cual resuelva un problema de vital importancia en las comunicaciones modernas.
* Desarrollar el trabajo en grupo, la capacidad de compartir conocimientos y discutir ideas, para afrontar un nuevo problema en común y entregar una solución bajo un acuerdo de todas las partes del grupo.
* Investigar un tema ajeno a nuestros conocimientos, y ser capaces de aplicar conocimientos adquiridos para poder explicar en términos no tan técnicos (en este caso de la asignatura) a personas que no conozcan o hayan trabajado con la idea propuesta.
* Implementar un modelo según la solución propuesta, que sea capaz de demostrar la importancia de nuestro proyecto.

## Introducción

La idea de este proyecto surge ante la necesidad de establecer conexión entre dos terminales, pero sin comunicación directa entre ellos, nuestro problema es:

“*Actualmente existen Plataformas Móviles que por diferentes motivos deben ser controladas a distancia y a la vez intercambiar información con un usuario remoto”*

Nuestra bosquejo de solución:

*“Sin tener información de las direcciones (ubicación) de la Plataforma Móvil y el usuario, se plantea instalar un servidor intermedio con IP Fija para establecer la comunicación. Por medio de sockets TCP se implementará la interfaz, donde el cliente envía información al servidor y éste la reenvía a la Plataforma Móvil”*

Hoy en día es necesario esto para monitorear equipos sin necesidad de presencia humana o que faciliten este trabajo, además las máquinas que se conectan a internet ya no son sólo computadores, existen sistemas de alarmas, GPS, celulares, televisores y otra gran cantidad de dispositivos, por ello es importante contar con servidores personalizados. Nuestro proyecto consistirá en cómo crear una interfaz capaz de actuar como servidor personalizado, al cual se conectarán dos equipos, un computador y un dispositivo electrónico (o otros usuarios), y sobre esta plataforma podrán interactuar bajo un modelo de conexión full-dúplex, donde podamos recibir información al mismo tiempo que la enviamos, en ambos sentidos.

Crearemos una interfaz de comunicación creando un servidor al cual estarán conectados ambos hosts (modelo cliente-servidor), se programarán sockets asociados a puertos específicos para la aplicación que vamos a utilizar, se trabajará sobre el modelo TCP/IP aprendido durante la cátedra, asegurando control de flujo y congestión para entregar fiabilidad en la comunicación la cual es de suma importancia para ambos terminales en este caso.

Algunos problemas que emergen sin un adecuado control de flujo, es que el usuario podría enviar información hacia otro usuario, y si no llega una respuesta, no podríamos identificar si el problema proviene del servidor o del otro usuario (por ejemplo, que no sepa la respuesta a la petición). O como priorizar envíos simultáneos de información hacia el servidor, repartición de memoria, que ocurre si un usuario satura al servidor con mensajes, etc.

## ARQUITECTURA DEL SERVIDOR

Los sistemas terminales son llamados anfitriones (Host), los cuales ejecutan los aplicaciones (Software) tanto de Servidor como Cliente.

Los computadores conectados a Internet son Sistemas Terminales, ya que están situados en el extremo de la red. Existe un número creciente de dispositivos alternativos que funcionan como Sistemas terminales, como lo son los clientes ligeros o aparatos domésticos, para este proyecto utilizamos la idea de que uno de los Sistemas terminales a utilizar corresponde a la categoría antes mencionada siendo una Plataforma Móvil que puede tener diferente IP.

El cliente es un programa que se ejecuta en un sistema terminal, el cual solicita y recibe un servicio de un programa servidor.

**Programación Sockets**

Como se vio en el ramo un socket es una puerta entre el proceso aplicación y el protocolo de transporte de extremo a extremo, son locales al host y son creados por cada aplicación, con la función de ser una interfaz a través de la cual el proceso aplicación puede recibir y enviar mensajes desde/hacia otro proceso aplicación.

La interfaz de sockets facilita la comunicación entre procesos, programando dicha comunicación de manera similar a como se maneja cualquier dispositivo de entrada salida, ya que independiza la localización de los procesos.

Un Socket permite implementar una arquitectura cliente/servidor para la que principalmente podemos mencionar las necesidades básicas para su implementación en TCP/IP:

- Un par de Direcciones IP de los host.

- Un par de número de puerto que representa al proceso corriendo en cada host.

Para el proceso de programación de los sockets existen dos tipos de aplicaciones cliente-servidor que se pueden generar:

- Estándar RFC: Cuando se desea crear una aplicación que funcione con algún estándar definido por RFC.

- Aplicaciones propietarias: están tiene la característica de que no se rigen por algún RFC existente y se debe tener cuidado en no utilizar puertos definidos en RFC.

Además para la programación de los socket se debe elegir si usar el protocolo TCP u UDP.

**¿Qué son los procesos y los threads?**

Los threads o hebras, permite a una aplicación realizar varias tareas a la red, la aplicación de ellas aparecen en el código del programa explícitamente en:

La definición un poco más formal de estos es:

*“Los distintos hilos de ejecución comparten una serie de recursos tales como el espacio de memoria, los archivos abiertos, situación de autenticación, etc. Esta técnica permite simplificar el diseño de una aplicación que debe llevar a cabo distintas funciones simultáneamente.* *Un hilo es básicamente una tarea que puede ser ejecutada en paralelo con otra tarea”.*

Entonces nosotros podemos asignar prioridad a un equipo, el cual podría ser “*Usuario principal*”para que tenga prioridad en el envío de datos por sobre el resto (así evitamos un problema de congestión en el buffer del servidor), o asignar por ejemplo, una tarea específica a cada usuario, considerando que pueda hacer más efectivo el uso de un gran dispositivo (el uso de varios micro-controladores por ejemplo, donde podríamos asignar cada uno a distintos usuarios pero todos dirigidos a un equipo común).

Cada thread tendrá asignado un sub-proceso, el cual por definición conforma un proceso, este último será quien le entregará la respectiva orden al dispositivo, el cual responderá a través de otra hebra con un mensaje al servidor, este último detectará que se trata de una hebra de respuesta desde el dispositivo, y tendrá que enviar como respuesta al respectivo usuario, esperando al mismo tiempo en otra hebra una oportuna petición de este usuario. Esto se diseña por igual para todos los usuarios, lo que variará para cada uno será la programación de la hebra específica , ya que el usuario con mayor prioridad no poseerá el mismo código que otro usuario, o a lo mejor tendrá asignado una hebra que relaciona una tarea totalmente distinta.

Es importante entender que este concepto de hebra, nos permite una comunicación full-dúplex, ya que cada una de ellas se pueden programar de distinta forma.

**Sockets a través de Python**

Para realizar nuestra tarea se utilizo el lenguaje Python ya que proporciona un marco para programar servidores de red que simplifica gran parte de la programación del socket propiamente tal y nos permite concentrarnos en la creación de los hilos, tanto plataforma como usuario. Además se piensa que resulta provechoso para el aprendizaje y experiencia mostrar un ejemplo en otro lenguaje de programación, utilizando otra alternativa para la creación de sockets.

El marco (framework) a utilizar lleva por nombre *SocketServer* el cual es un modulo que define diferentes clases servidor que se pueden utilizar en este marco, las cuales son: TCPServer, UDPServer, UnixStreamServer y UnixDatagramServer, estas dos últimas deben ser utilizadas en plataformas Unix.

La clase a ejemplificar es TCPServer la cual requiere sincronismo, esto es que cada solicitud realizada por algún cliente deba ser procesada antes de poder responder a una segunda, lo cual puede ser inadecuado si las solicitudes tardan mucho en ser procesadas, el modulo SocketServer permite solucionar este problema creando procesos independientes para manejar cada solicitud.

Para crear el servidor se requieren los siguientes pasos:

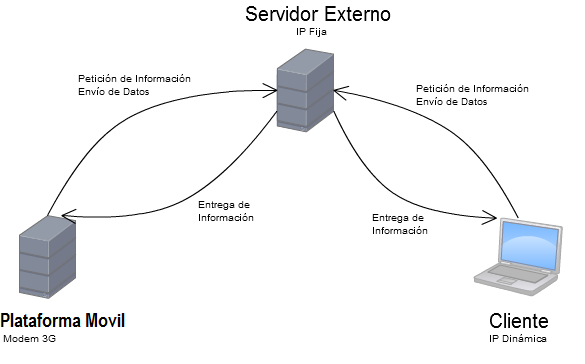
1 . Primero hay que instanciar una clase que maneje las solicitudes de conexión de los clientes, dentro de esta se definirán las funciones para nuestros dos clientes, la plataforma y el usuario que tiene la opción de solicitar datos o enviar parámetros para el movimiento del dispositivo.

2. Se declara la clase que maneja la hebra servidor y se asigna la característica de servidor asíncrono. Se utiliza la clase "ThreadingMixIn" que añade funcionalidad al servidor TCP de forma que cada nueva conexión crea un nuevo hilo.

3 . Luego se debe instanciar una de las clases servidores, pasando como parámetro el puerto y la dirección IP, la cual en nuestro caso es un Servidor TCP.

4. Por último se debe especificar que el servidor se mantenga a la espera de conexiones entrantes y que se mantenga encendido.

A continuación se presenta una diagrama de la interfaz de comunicación a utilizar.



1. Diagrama de Interfaz de Comunicación

**Envió de datos**

El usuario y la plataforma deben enviar los datos en un formato especificado para que puedan mantener un entendimiento y realizar las peticiones en forma adecuada, sin que se presenten acciones no solicitada. Se define un pequeño protocolo de capa de aplicación para el envió de datos en nuestro sistema:

Los datos recibidos por el servidor debe ser una cadena de caracteres de la forma:

*DISPOSITIVO*:*ACTUADOR(SENSOR):TIPO\_MOV*:*VELOCIDAD(SENTIDO)*

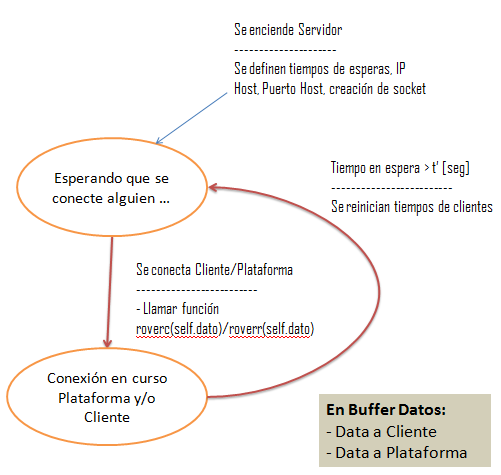
Dispositivo: Indica el dispositivo que se está conectando (PLATAFORMA o USUARIO). De esta forma permite al Servidor reconocer quien se está conectando y por lo tanto a cuál de las dos funciones cliente llamar. Estas funciones implementan las características de comunicación tanto del usuario como de la plataforma

Para los demás elementos del mensaje , dependiendo de quién se haya conectado, cada una de éstas, indica las opciones que serán usadas luego de que los datos lleguen a su respectivo destino.

Como se ve, en la etapa del servidor solo se usa la primer elemento de la cadena de caracteres, lo que tiene una similitud con la forma en que trabajan los protocolos en Internet ya que hace uso de una parte del "paquete" de datos y el resto lo envía en forma normal.

La inclusión de herramientas de programación puede facilitar la ejecución de múltiples tareas, en este proyecto por ello se realiza una programación orientada a objetos.

Para poder entender mejor como el servidor pasa de un estado a otro, utilizamos diagrama de estados como los analizados en clases.



1. Diagrama de estados Servidor

El servidor espera que la plataforma o el cliente se conecta, cuando uno de los dos se conecta este almacena los datos (si es que los hay) en un buffer destinado a la otra plataforma

Entonces cuando una de las dos se conecta nuevamente este envía los datos almacenados al cliente que corresponde. De esta forma el Servidor funciona como una nube donde cada plataforma/cliente consulta si hay datos destinados para ellas.

Las funciones *roverc(self.dato)* y *roverr(self.dato)* permiten realizar el envió a buffer de los datos recibidos y enviar datos en caso de que los halla al cliente que los solicita.

## Conclusiones

* Como grupo profundizamos ciertos conceptos de la asignatura, que en principio no estaban tan claros, además de aprender elementos extras relacionados a la asignatura y que no fueron evaluados en ella, por ejemplo programación en Java de Sockets, programación de servidor en lenguaje Python, el concepto de hebras y que lo aprendido en esta asignatura es una base para mundo actual el cual funciona con otros conceptos basados en lo ya aprendido.
* Se observo que existen elementos relacionados a la hora de afrontar diferentes tipos de problemas, principalmente en sus soluciones, donde teniendo una idea de cómo "solucionar" un problema en un ámbito de la Ingeniería se puede llevar esta forma de solución a otro problema parecido en otro ámbito, por ejemplo en la comunicación entre dos clientes, donde uno no conoce la ubicación del otro (IP) es posible introducir un servidor entre ellos y adoptar una arquitectura cliente/servidor (como la estudiada en clases) entre cada cliente por separado donde solo sea necesario que los respectivos clientes consulten si hay datos esperando para alguno de ellos.
* Se desarrollo una aplicación Servidor con las características básicas, por medio de hebras y elementos de la programación orientada a objetos, para acercarnos a desarrollos más elaborados. Principalmente se ideo un ejemplo de utilización en un ambiente real con una plataforma móvil, con lo cual se deben tener consideraciones especiales que sin elementos estudiados en el ramo no sería posible llevar a cabo, como por ejemplo el uso de IP dinámicas y el hecho de que si nuestra plataforma tiene una IP diferente, la próxima vez se desee conectar a ella tendremos problemas, de esta forma se obtiene la idea de que se necesita un Servidor con una IP fija para poder conectarse cuando uno lo desee.

## Referencias

* http://mundogeek.net/tutorial-python/ Tutorial Python
* Redes de Computadores-James F.Kurose, Keith w. Ross
* http://docs.python.org/library/socketserver.html SocketServer en Python
* http://www.wikipedia.com

## ANEXOS

Anexo 1: Código en Lenguaje de Programación Python del Servidor TCP/IP

'''

Created on 07-09-2011

@author: cristobalbl

'''

import socket

import threading

import SocketServer

import time

## Guarda la dirección de movimiento que se le envía a ROVER

data\_rover = ""

data\_rover\_aux = ""

## Guarda los datos que se le envía al cliente

data\_cliente = ""

data\_cliente\_aux = ""

## Tiempo de ROVER tiempo muerto, tiempo desde que se cerró la conexión

tiempor = 0

## Tiempo del cliente tiempo muerto, tiempo desde que se cerró la conexión

tiempoc = 0

## estado de la conexión ROVER

rconnected = False

## estado de la conexión cliente

cconnected = False

#CLASE MANEJADOR DE PETICIONES

class ThreadedTCPRequestHandler(SocketServer.BaseRequestHandler):

def handle(self):

global cconnected, rconnected, tiempoc,tiempor

# Datos recibidos de la forma DISPOSITIVO:ACTUADOR(SENSOR),TIPO\_MOV,OPC

data = self.request.recv(1024)

print "Alguien se conecto"

print "datos recibidos: " + data

self.datos = ""

self.quien = ""

self.data\_splitter(data)

# Se detecta dispositivo conectado

if self.quien == 'ROVER':

if not rconnected:

rconnected= True

tiempor = time.time()

print "ROVER conectado"

self.roverr(self.datos)

elif self.quien == 'CLIENTE':

if not cconnected:

cconnected = True

print "Cliente conectado"

tiempoc = time.time()

self.roverc(self.datos)

else:

pass

# Divide los datos recibidos en (quien lo envía, acción a realizar)

def data\_splitter(self,data):

self.quien, self.datos = data.split(":",1)

if self.quien == '1':

self.quien = 'ROVER'

elif self.quien == '2':

self.quien = 'CLIENTE'

## Función a llamar cada vez que se conecta cliente 1

def roverr(self,dato):

global data\_cliente

global data\_rover\_aux

if not data\_rover == data\_rover\_aux:

self.request.send(data\_rover)

print "Enviado a ROVER: " + data\_rover

data\_rover\_aux = data\_rover

else:

self.request.send("esperando datos...")

if not dato.count("conectando"):

data\_cliente = dato

self.request.send("datos asignados a data\_cliente")

print data\_cliente

# Función a llamar cada vez que se conecta cliente 2

def roverc(self,dato):

global data\_rover

global data\_cliente\_aux

print "CLIENTE conectado"

#Si cambia datos a cliente (vienen de ROVER) se envían a cliente

if not data\_cliente == data\_cliente\_aux:

self.request.send(data\_cliente)

print "Enviado a CLIENTE: " + data\_rover

data\_cliente\_aux = data\_cliente

else:

self.request.send("Sin datos desde ROVER recibidos")

#El dato enviado por el cliente, pasa a ser igual a una variable data\_rover para ser enviada a ROVER

if not dato.count("conectando"):

data\_rover = dato

self.request.send("datos asignados a data\_rover")

print data\_rover

# Permite el uso de Hebras

class ThreadedTCPServer(SocketServer.ThreadingMixIn, SocketServer.TCPServer):

pass

if \_\_name\_\_ == "\_\_main\_\_":

global cconected, rconected

HOST, PORT = "localhost", 1234

#Creando el servidor

server = ThreadedTCPServer((HOST, PORT), ThreadedTCPRequestHandler)

server\_thread = threading.Thread(target=server.serve\_forever)

server\_thread.setDaemon(True)

server\_thread.start()

print "Servidor corriendo: ", server\_thread.getName()

#Loop del Servidor (Maquina de estados)

while True:

if cconnected:

if time.time() - tiempoc > 2:

cconnected = False

data\_rover = ""

print "Cliente desconectado"

if rconnected:

if time.time() - tiempor > 2:

rconnected = False

data\_cliente = ""

print "Rover desconectado"

server.shutdown()