UNR Universidad Nacional de Rosario

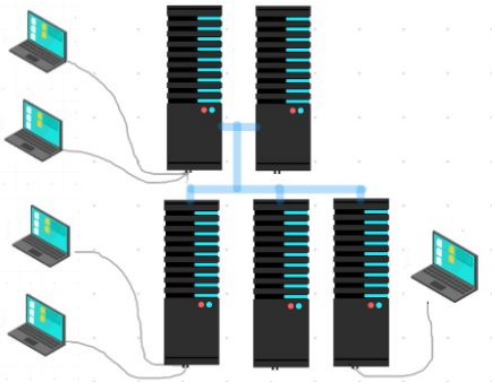
FCEIA

Trabajo práctico 2 - Sistemas Operativos 1

Alexis Emanuel Barraud
Leonid Chanco Castillo

## Arquitectura del servidor distribuido

La implementación consiste con varias computadoras conectadas conectadas entre sí, las cuales se representan con máquinas virtuales de Erlang (Nodo), donde cada una de ellas está escuchando por un puerto las peticiones de los clientes.



Cada Nodo sabe cual de todos los nodos conectados a él tiene menor carga, al recibir una petición de un cliente, este se envía a procesar en dicho nodo.

Si bien los clientes pueden tener participación con los juegos o procesos de los otros nodos los nodos tienen una estructura individual, si uno de ellos se cae, no afecta de manera significativa al servidor, solo se perdera la informacion del nodo implicado.

## Protocolo de comunicación

La comunicación con el servidor se realiza mediante el envío de strings sobre un socket TCP con la forma “CMD cmdid op1 op2 …” descritas en el enunciado del tp.
La comunicación da lugar a realizar el cliente de forma independiente, con cualquier otro lenguaje que soporte el protocolo anterior, en nuestro caso decidimos también realizarlo con Erlang.

## Peticiones de los clientes

Cuando un cliente se conecta se crea una nueva función “psocket” el cual se encarga de recibir las peticiones de los usuarios.
Por cada petición que realiza un cliente se asocia a un “pcomad” una función que procesa la petición en el nodo de menor carga.

## Manejo de carga

Por cada Nodo servidor se implementa una función “Pstat” y un controlador de carga, el cual se encarga de pedir información sobre la carga a cada nodo conectados, esto preguntando a cada “pstat”, de tal forma que cada nodo sabe cual de todos los conectados a él tiene menor carga, así cuando se quiera ejecutar un “pcomand” se tiene a disposición el nodo de menor carga. La actualización se realiza en nuestro caso cada 5 segundos.

## Registro de los usuarios

Para representar un usuario decidimos utilizar un record llamado **userMind** donde el mismo es almacenado en un mapa en el nodo servidor donde fue creado:

```
userMind{
        nombre,
        pid }
```

donde:

- **nombre** : representa el nombre del usuario.
- **pid** : representa el PID del proceso "conciencia" ya que cada vez que se agrega un usuario al servidor en el nodo correspondiente se crea dicho proceso el cual se encarga de estar pendiente si el cliente cerró la conexión o no, puesto que en caso de que el cliente haya cerrado la conexion y ademas si este estaba jugando una partida del tateti el servidor será el encargado de avisar a los observadores y al contrincante que dicho usuario ha abandonado la partida.

## Registro de juego

Para representar un juego decidimos utilizar un record llamado **game** donde el mismo es almacenado en un mapa en el nodo servidor donde fue creado. Dicho record contiene los siguientes campos:

```
game {
        idGlobalGame,
        name,
        status,
        pidControl}
```

en donde:

- idGlobalGame : representa el id del juego, es decir, el identificador del juego creado.
- name : representa el nombre del juego.
- status : el cual representa el estado del juego ya que puede estar activo (caso en donde ya la partida posee las personas necesarias para poder jugar) o en espera (caso en donde falta que alguna persona acepte el juego para poder comenzar a jugar).
- pidControl : almacena el PID del juego lanzado.

Además cada vez que un juego es registrado se lanza un hilo joystick el cual será encargado de interactuar directamente con el juego y realizar las operaciones básicas que son requeridas por el tp como unirse, observar, jugar, etc.

## MANEJO DE ERRORES

**Se desconecta un cliente:**

Cada cliente que se registra al servidor tiene una función “f conciencia” asociada a él, linkeado con cada juego donde en el que está participando como “observador o jugador”. Cuando un socket se desconecta, es capturado y enviado a una función “la Parca”, luego obtiene el nombre asociado al socket desconectado y envía un mensaje de desconexión a la función “f conciencia” para que esta finalice, de forma tal que todos los juegos linkeados a el tambien se enteren.

**Se cae un servidor (nodo):**

La información de los juegos y usuarios de el nodo implicado se pierde, y los usuarios conectados a él tienen la opción de conectarse a otro servidor.

**Proceso de un pcomand:**
Cuando un cliente realiza una petición, este se envía a procesar a un pcomand en el nodo de menor carga, si este nodo se rompe, se captura el error y se envía a procesar en el nodo de menor carga activo, así mientras tengamos nodos activos.

## Posibles mejoras

En la arquitectura del servidor se puede implementar un respaldo de información relevante, para cuando un nodo se cae, este se restaure en otro nodo con el mismo puerto, así los usuarios al reconectarse podrán seguir con la información restaurada.

Se puede crear más aplicaciones para los clientes, estos pueden ser más juegos o un chat, puesto que las funciones joystick se comunica con el tateti de nuestra implementación de forma independiente.

Mejorar la interfaz de usuario, con un lenguaje que soporte gráficos para el cliente.

Conectar una base de datos para guardar estados de partida o el registro de usuarios.

Para la inserción o búsqueda de usuarios o juegos podríamos usar estructuras de datos más eficientes.

## Proceso de ejecución

Archivos disponibles:

```
crlPstat.hrl crlCarga.hrl crlUserSock.hrl crlListGame.hrl crlJostick.hrl pcomands.hrl server.erl
tateti.erl cli.erl README.md
```

Los pasos para ejecutar el servidor se encuentran en el archivo README.md

Si se requiere hacer testeos en plena ejecución, se puede invocar, en la máquina donde está activo un servidor, funciones implementadas en los hrl.
Ejemplo:
server:usrGets(node()). retornara el mapa de todos los clientes registrados en dicho nodo
server:gsGets(node()).  retornara el mapa de todos los juegos registrados en dicho nodo