Trabajo Práctico Final

Sistemas Operativos I

Florencia Rovere Guido De Luca Tomás Lopez

08/03/2017

Introducción

Se presenta un servidor de Ta-Te-Ti distribuido escrito en Erlang, mediante el cual un jugador registrado podrá inicializar una partida, aceptar una partida de otro jugador o ingresar a una partida como observador desde el momento en que se crea la misma. Para que funcione de manera distribuida, se deberán correr varios servidores y conectarlos entre sí, como se explicará más adelante. Luego, el cliente podrá jugar viendo a los servidores como si fueran uno solo.

Implementación

Servidor

Módulos utilizados

Para establecer la conexión con el cliente utilizamos el módulo `gen_tcp`; y utilizamos el módulo `net_kernel` para poder inicializar al servidor como un nodo dentro de un sistema distribuido y para luego poder conectarnos con el resto. Explicamos estos pasos en la sección sobre cómo compilar. Tanto para el registro de usuarios como para el registro de partidas, utilizamos el módulo `global`. Esto nos permitió registrarprocesos como `user/2` o `match/4` de forma tal que sean visibles por todos los nodos, garantizándonos de esta forma, una comunicación interna bien delimitada y clara.

Funcionamiento

En cada nodo servidor tendremos un hilo corriendo `pbalance/2`, otro corriendo `stat/0` y otro `match_adm/1`. Los procesos `pbalance/2` y `stat/0` son los encargados de mantener una correcta distribución de cargas entre los nodos, mientras que el proceso `match_adm/1` es el que se encargará de administrar las partidas. En caso de cambiar algún parámetro en cualquiera de estos procesos, deberá informarse la actualización en todos los nodos. Por ejemplo, cuando `stat/0` envía las estadísticas a `pbalance/2`:

```
stat() ->
...
pid_balance!{stat, TotalTasks, node()},
lists:map(fun(X) -> {pid_balance,X}!{stat, TotalTasks,node()} end, nodes()),
...
```

Una vez lanzados los hilos mencionados, el servidor concluye su inicialización llamando a `dispatcher/3`. A partir de ahora, el servidor se encargará de aceptar las conexiones entrantes y, por cada una de ellas, creará dos hilos nuevos que correrán los procesos encargados de comunicarse con el cliente durante todo el tiempo que permanezca conectado: `psocket/5` y `listener/1`. Cada vez que un cliente introduzca un comando, `psocket/5` lanzará un proceso `pcommand/6` en un nuevo hilo. Notar que gracias a `pbalance/5`, este nuevo hilo será creado en el servidor menos congestionado. Una vez procesada la información recibida, `psocket/5` enviará la respuesta correspondiente al cliente. La tarea de `listener/1` consiste en enviarle información al cliente, que no surge a partir de la introducción de un comando. Un ejemplo de esto es la actualización del tablero: cada vez que el usuario deba ser notificado de un cambio en el tablero, se le enviará la información a través de este proceso. La funcionalidad

de `listener/1` no se limita solamente a esto, pero sí es el ejemplo más sencillo de observar.

La representación de cada partida se realiza mediante un proceso globalmente visible como mencionamos arriba. La misma se inicia ejecutando el proceso `match/4`, que representa al juego en una instancia de "lobby". Es decir, en esta instancia sólo hay un jugador en la sala que se encuentra a la espera de un contrincante. Una vez que otro jugador accede a la sala, se ejecuta el proceso `match_initialized/6` quien se encargará de administrar las jugadas que se realicen hasta el final de la partida. Cabe destacar que al correr `match_initialized/6` sobre el mismo hilo que `match/4`, el PID no cambia y el átomo que se utilizó para el registro global sigue siendo una forma válida para que otro proceso se comunique con la partida.

Cliente

Para iniciar un cliente, lanzamos el proceso `start/2`, al cual debemos especificarle en qué host y en qué puerto se encuentra el servidor. Luego, utilizando la función `connect/3` provista por el módulo `gen_tcp` establecemos la conexión. A partir de este momento, dos procesos corriendo en dos hilos distintos y comunicándose entre sí, serán los pilares de nuestro cliente: `console/2` y `listener/2`.

El primero se limita a mostrar un prompt en pantalla; y una vez que un comando fue ingresado, enviarlo hacia el servidor, quien se encargará de verificar que lo ingresado tenga -o no- sentido. Finalmente, se queda esperando que `listener/2` le indique si debe continuar o cerrar la conexión.

Simultáneamente, `listener/2` es el encargado de escuchar los mensajes provenientes del servidor, interpretarlos, y en función de eso realizar la acción designada.

Comandos

Las operaciones que puede realizar el cliente son:

- HELP: lista los posibles comandos a ingresar.
- LSG: lista las partidas registradas y su estado.
- BYE: cierra la conexión del cliente y abandona todas las partidas en la que participa.
- CON ⟨ nombre-de-usuario ⟩: conecta al usuario con el nombre especificado. Si dicho nombre ya está en uso, se le solicitará que ingrese otro.
- NEW ⟨ nombre-de-la-partida ⟩: crea una nueva partida con el nombre dado. En caso de estar en uso el nombre, se le pedirá al jugador que ingrese otro.
- ACC ⟨ nombre-de-la-partida ⟩: accede a la partida con el nombre indicado. Tanto si la partida no existe como si ya hay dos jugadores en la misma, se le enviará al jugador un mensaje de error con el motivo por el cual no pudo acceder.
- OBS ⟨ nombre-de-la-partida ⟩: el jugador comienza a observar una partida en curso. Puede que el nombre no sea válido y en dicho caso, se notificará la situación.

- LEA ⟨ nombre-de-la-partida ⟩: se utiliza para dejar de observar una sala. En caso de no existir la misma, se avisará al jugador.
- PLA ⟨ nombre-de-la-partida ⟩ ⟨ jugada-x ⟩ ⟨ jugada-y ⟩: realiza una jugada en la partida dada con las coordenadas (⟨ jugaga-x ⟩, ⟨ jugada-y ⟩), si es posible la misma. Si no, se enviará un mensaje al jugador con el motivo del error.
- PLA ⟨ nombre-de-la-partida ⟩ END: finaliza la partida especificada. En caso de no ser partícipe de la misma, no será válida la jugada.
- SAY ⟨ nombre-de-jugador ⟩ ⟨ diálogo ⟩: Envía el texto al jugador ingresado. Si no se encuentra el mismo, se notificará el error.

Cómo compilar

server.erl

Abrir una terminal de Erlang en la carpeta **src/** y ejecutar:

```
1> c(server).
2> server:start(nombre-del-servidor, puerto).
```

Luego, si queremos conectar dos servidores para que funcionen de manera distribuida:

```
1> c(server).
2> server:start(nombre-del-servidor-2, puerto-2).
3> net_kernel:connect('nombre-del-servidor@hostname').
```

client.erl

Abrir una terminal de Erlang en la carpeta **src/** y ejecutar:

```
1> c(client).
2> client:start({a,b,c,d}, puerto-del-servidor).
```

donde {a,b,c,d} representa la dirección IPv4 a.b.c.d en la que se encuentra corriendo alguno de los servidores.

También podemos utilizar la función `client:localstart/1` para iniciar clientes en servidores dentro del localhost:

```
1> c(client).
2> client:localstart(puerto-del-servidor).
```