POSTMORTEM – TALLER 6

En general el trabajo no nos ha ocupado una cantidad de tiempo grande porque el taller tenía un espectro de trabajo relativamente bajo y hemos sido capaces de resolver los problemas que han aparecido de forma ágil gracias a la experiencia obtenida en los talleres de *Tcp*.

Al principio fue complicado cambiar la forma de pensamiento respecto a *Tcp* y con esto el planteamiento inicial. Una vez conseguimos hacer la conexión inicial la única dificultad notable que encontramos fue la emisión de los mensajes de confirmación de mensajes críticos. Al principio teníamos la emisión de confirmación dentro del proceso del mensaje de entrada, pero no los enviaba por tanto lo que hemos hecho es guardarnos todos los mensajes de confirmación que tenemos que enviar y mandarlos después de procesar los mensajes de entrada (al final del *loop* principal).

Al principio teníamos los sockets a *blocking* pero vimos que al acabar el *loop* principal el programa no acababa porque el *thread* de recepción se quedaba bloqueado en el receive y no comprobaba si el programa había acabado. Por tanto, cambiamos los *sockets* modo *non-blocking* y controlamos el estado *partial*.

Hemos implementado un *ping* que lo utilizamos para controlar si los clientes siguen conectados (a los 5 segundos de no recibir respuesta determinamos que ha habido una desconexión y se notifica a los demás). Además, utilizamos este mensaje para mandar una actualización de mundo para que se sincronicen todos los clientes (hemos implementado esta funcionalidad porque para nuestra practica nos hará falta). Los clientes también aprovechan este *ping* para determinar si el servidor sigue conectado o deben apararse.

Ya utilizamos una *GameLib* donde tenemos una clase cliente, una clase *evento* y un *header* para funciones de utilidades.

Protocolo de comunicación

|  |  |  |
| --- | --- | --- |
| **Mensaje** | **Sentido** | **Descripción** |
| HELLO\_ | C🡪S | Petición de conexión de un cliente al servidor. |
| WELCOME\_<id>\_  <sizeClients>\_[<clientNId>  \_<clientNx>\_<clientNy>] | S🡪C | El servidor confirma el establecimiento de conexión y envía la información del ID y posición de los jugadores que había conectados antes junto con la posición de este cliente en particular. |
| NEWPLAYER\_<id>\_<x>  \_<y> | S🡪C | Cuando un jugador nuevo se conecta el servidor manda un mensaje a todos los clientes ya conectados informándoles del ID de este jugador y de su posición en el mapa. |
| CRITICAL\_<id> | C🡪S & S🡪C | Marca que es un mensaje crítico y que se tendrá que tratar de una forma especial. Este mensaje necesitará una confirmación de recepción.  El *id* que lleva es para diferenciar el mensaje crítico para facilitar la confirmación. |

|  |  |  |
| --- | --- | --- |
| ACK\_<id>\_<msgID> | C🡪S | Confirma la recepción de un mensaje crítico.  Envía el id propio del cliente para que el servidor procese quien manda el mensaje y el id del mensaje crítico para poder saber qué mensaje es el que está confirmando. |
| PING\_<sizePlayers>\_  [<idN>\_<Nx>\_<Ny>] | S🡪C | Mensaje que envía el servidor para controlar que los clientes sigan conectados y aprovechamos para enviar el estado del mundo y sincronizar todos los clientes. |
| ACKPING\_<id> | C🡪S | Confirma la recepción del mensaje de ping. Este lo utilizamos para controlar que los clientes siguen conectados. Los clientes mandan su ID para que el servidor sepa quiénes son rápidamente. |
| DC\_<id> | S🡪C | El servidor manda un aviso de desconexión de un jugador. El id marca que jugador es.  Todos los clientes eliminarán a este jugador de sus listas. |

Utilización de la aplicación