PROGRAMACION DE REDES

Gonzalo Kurin 19248 - Nicolás Damiani 192489

UNIVERSIDAD ORT Facultad de Ingeniería

Table of Contents

Alcance del Sistema **3**

Funcionamiento **3**

Arquitectura **3**

Justificación de diseño **5**

Diagrama de Paquetes **7**

Diagramas de Clases **8**

- **Diagrama del paquete Entities** **8**
- **Diagrama del paquete Server** **9**
- **Diagrama del paquete Client** **9**
- **Diagrama del paquete Protocols** **10**
- **Diagrama del paquete Exceptions** **11**

Diagramas de Secuencia **12**

- **Registro de Usuario** **12**
- **Movimiento (análogo con ataque)** **13**
- **Subir archivo** **14**

Alcance del Sistema

La solución implementada contempla en su alcance toda la funcionalidad requerida por medio de la letra. Se implementaron dos aplicaciones, una para el servidor y otra para los clientes. En caso del cliente, se implementó un menú principal para acceder a las funcionalidades del sistema. En la aplicación cliente se pueden realizar las siguientes funciones:

- Conectarse y desconectarse del servidor.
- Dar de alta un jugador.
- Conectarse al juego.
- Unirse a una partida activa.
- Seleccionar un rol al unirse a una partida.
- Realizar acciones (mover o atacar).
- Notificar el resultado de la partida al finalizar.

Por otro lado, las funciones del servidor son las siguientes:

- Aceptar pedidos de conexión de un cliente.
- Dar de alta y validar un pedido de alta de un jugador.
- Mostrar jugadores registrados.
- Mostrar jugadores conectados actualmente.
- Iniciar partida.
- Finalizar partida.
- Permitir a un jugador unirse a la partida activa.
- Mostrar resultado de una partida.

Funcionamiento

Para correr el sistema se debe dirigir a la carpeta Releases dentro de la carpeta de entrega, ahí van a haber dos carpetas, ReleaseClient y ReleaseServer.

Para correr la aplicación cliente se debe entrar a Release Client, cambiar la ip en el archivo de configuración y correr el archivo Client.exe y para correr la aplicación servidor, entrar a ReleaseServer, cambiar la ip en el archivo de configuración y correr el archivo Server.exe.

Arquitectura

La arquitectura implementada para el desarrollo del sistema requerido es una arquitectura cliente-servidor, donde el cliente es quien realiza peticiones al servidor, y este último le

responde.

Del lado del servidor, se tiene un socket que va a estar recibiendo pedidos de conexión por parte de los clientes. Cuando llega un nuevo pedido de conexión, se crean los *threads* necesarios para aceptarla, manejarla y procesar las peticiones que haga el cliente.

Del lado del cliente, se tiene un *socket* que se va a conectar al servidor y mediante el cual se va a intercambiar la información durante la conexión. Este *socket* se conecta al servidor y luego, mientras el cliente esté conectado, se itera sobre la siguiente secuencia:

- Se solicita un comando por consola.
- Se procesa el comando (se arma un paquete de protocolo) y se envía el request al servidor.
- Se recibe la respuesta.
- Se procesa la respuesta.

Si el comando ingresado por consola es el comando para cerrar sesión, se le hace *close* al *socket* del cliente (para enviar y recibir) y se lo cierra, liberando el recurso. Los *sockets* de los clientes se mantienen siempre conectados mientras el cliente se esté ejecutando.

Justificación de diseño

La solución propuesta está formada por cinco proyectos: Exceptions, Protocols, Client, Entities y Server. Cada uno de ellos contiene información relevante según lo describe su nombre y todos resultaron necesarios para cumplir con lo solicitado. A continuación, se pasará a detallar los ítems más relevantes de cada uno.

En DataEntities se definen las entidades necesarias para representar la lógica a nivel de dominio. En este proyecto se encuentran tres clases de importancia: Character, Match y User. En la clase Character se encuentra principalmente la información relevante al personaje que el usuario elige, así como su estado en el juego. Es decir, las constantes de vida y ataque, como también la vida y ataque actual del mismo en el momento dado del juego. La clase User se asocia con un Character, y a su vez tiene más información, como su estado de conexión, información del usuario y el turno en el que se encuentra. Dentro de la clase Match se encuentra toda la lógica del juego. Es decir, se controlan los turnos, los movimientos y los ataques, realizando verificaciones que permiten determinar si la acción es válida como también si existen ganadores o si la partida finalizó. A su vez en esta clase es donde se realiza gran parte del control de concurrencia. Aquí es donde se controla que dos usuarios no se muevan, ni ataquen, ni ganen a la misma vez, haciendo uso de locks que permitan bloquear las acciones de uno sin que intervengan a la misma vez que otro. Finalmente, en este paquete se encuentran clases que simplemente definen enums como lo son: CharacterType y ActionType.

En Server y en Client se encuentran todos los métodos y atributos necesarios para poder cumplir con los requerimientos especificados. Del lado del servidor, se aceptan pedidos de conexiones de clientes y se valida el correcto ingreso del usuario para que no existan dos con el mismo nombre. También se evalúan las peticiones recibidas de los clientes y se ejecutan las respuestas según corresponda. Lo mismo sucede del lado del cliente para poder así lograr que se realice toda la funcionalidad descrita en la sección de alcance de la solución.

Como explicamos anteriormente al describir el proyecto Match, el sistema permite que haya más de un usuario conectado al mismo tiempo interactuando con el servidor. Es por esto, que se debió tener ciertas consideraciones al momento de acceder a objetos comunes para evitar que se acceda a los mismos desde más de un thread y no aparezcan errores lógicos entre las ejecuciones de los mismos a la misma vez. En estos casos se utilizó el lock a través de un objeto compartido, para poder asegurar la mutua exclusión de los recursos.

La comunicación entre el cliente y el servidor se hace mediante el uso de un protocolo. El mismo se encuentra definido en el proyecto Protocol y permite el intercambio de información de manera uniforme. Para lograr esto, el protocolo codifica y decodifica la información transmitida, a partir de códigos y comandos definidos en la clase ProtocolConstants. A continuación, la descripción del formato general de la trama utilizada:

Nombre del Campo	Tipo de Mensaje	Código de Mensaje	Largo de la Data	Data
Valores	REQUEST/RESPONSE	LOGIN/ AVATAR_UPLOAD/ SELECT_CHARACTER/ JOIN_MATCH/ MOVEMENT/ ATTACK/ END_OF_MATCH/ ERROR	Integer	Variable
Largo	3 bytes	2 bytes	8 bytes	Variable

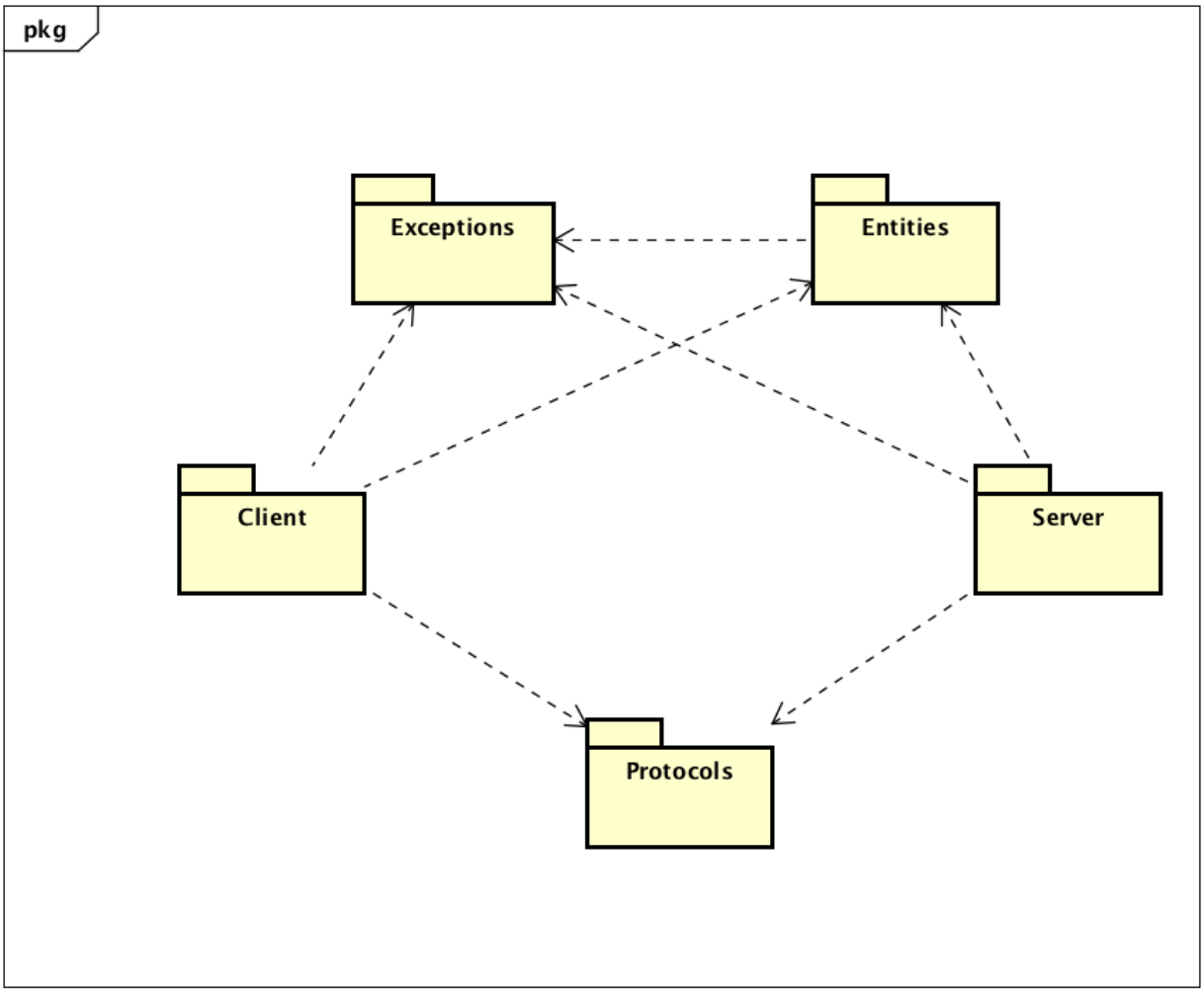
El tipo de mensaje representado en la trama está formado por la palabra “RES” o “REQ”, dependiendo de si es una petición o una respuesta. El código de mensaje se obtiene a partir de las constantes definidas en ProtocolConstants, y representa la petición que se quiere realizar. El largo de la data representa el tamaño de información a transmitir. Finalmente, en la Data se transmite la información. Este es el protocolo que se utiliza para todas las acciones, a excepción del envío de enviar el avatar al servidor. Al ser este un archivo con un tamaño grande se decidió agregar un campo de 3 bytes que indique la cantidad de partes que se envían. Por lo tanto, en la primera solicitud se envía esta información, agregando una parte del archivo. Las restantes partes del archivo son enviadas de manera continua, y con la información presentada en la primera solicitud esto puede ser controlado por el servidor y poder unir todas las partes para convertir a un archivo.

Manejo de errores

En el sistema se realiza el manejo de errores de dos formas:

- Mensajes de error. Cuando se detectan errores al ingresar datos, como el ingreso de una opción de menú inválida, se muestran mensajes de error en la consola para informar el problema ocurrido y se le solicita al usuario que vuelva a ingresar los datos.
- Excepciones. En muchos casos se utilizaron bloques try catch para capturar excepciones y manejarlas. Un ejemplo claro de estas situaciones se ve en el ServerLogic donde se manejan distintas situaciones, y a partir de la excepción atrapada se envía a la aplicación del cliente mediante una respuesta con código ERROR (99). Además, también se atrapan las excepciones de finalización de conexión y otras situaciones para controlar el correcto flujo de la aplicación, y que todo siga funcionando correctamente.

Diagrama de Paquetes



Diagramas de Clases

Diagrama del paquete Entities

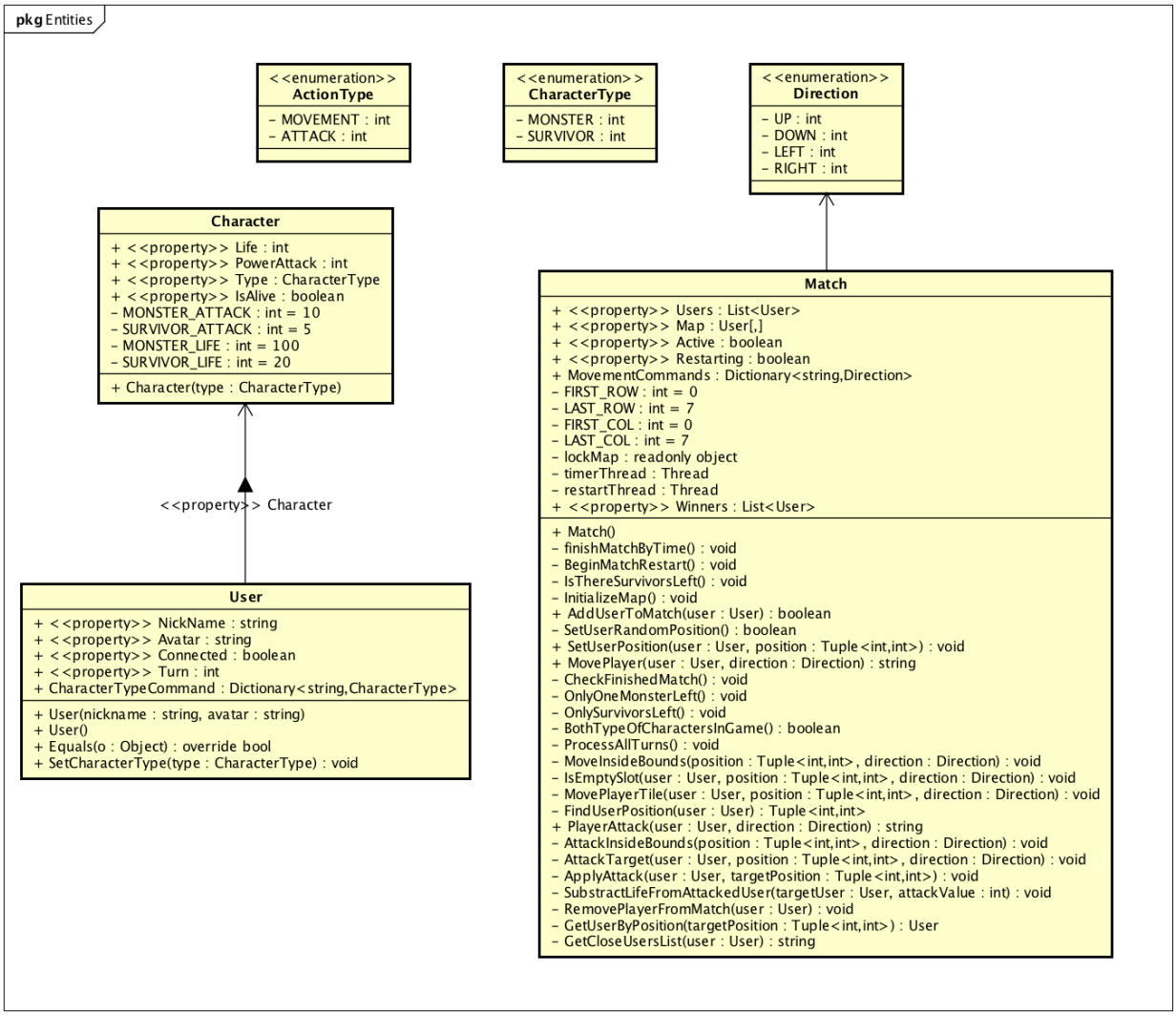


Diagrama del paquete Server

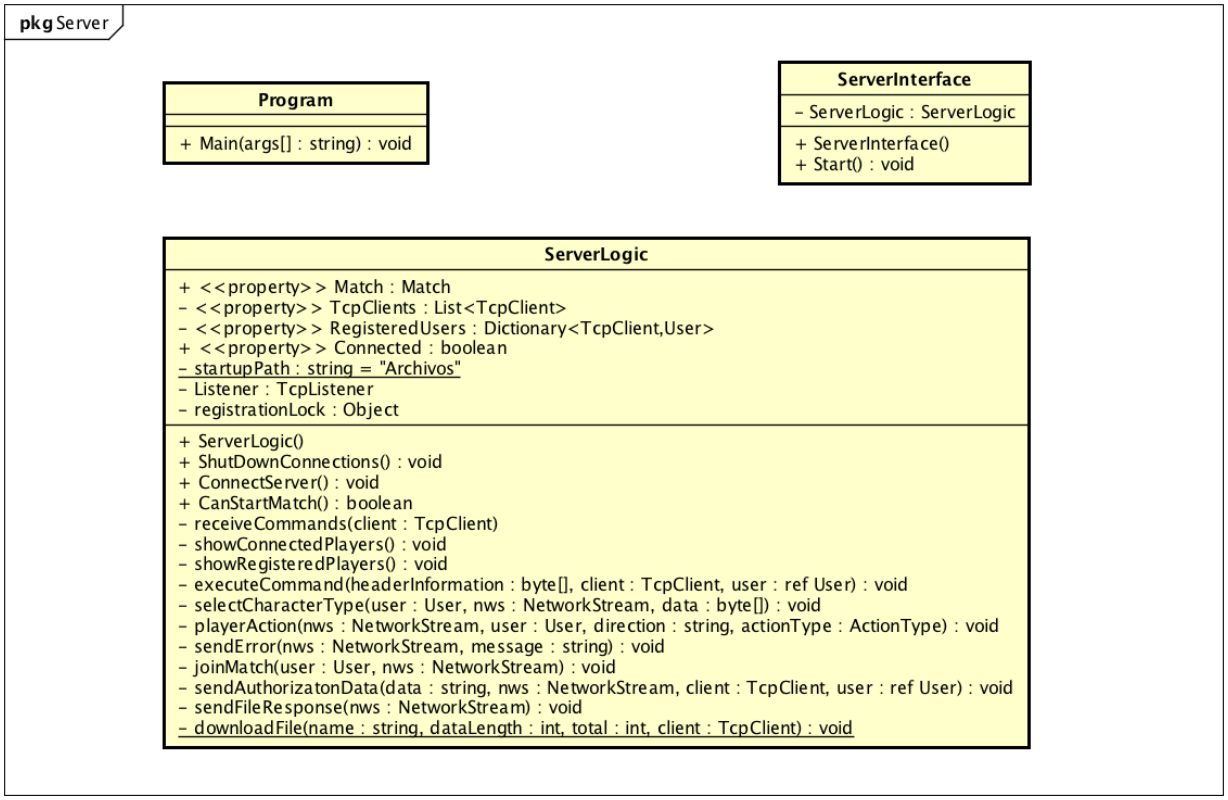


Diagrama del paquete Client

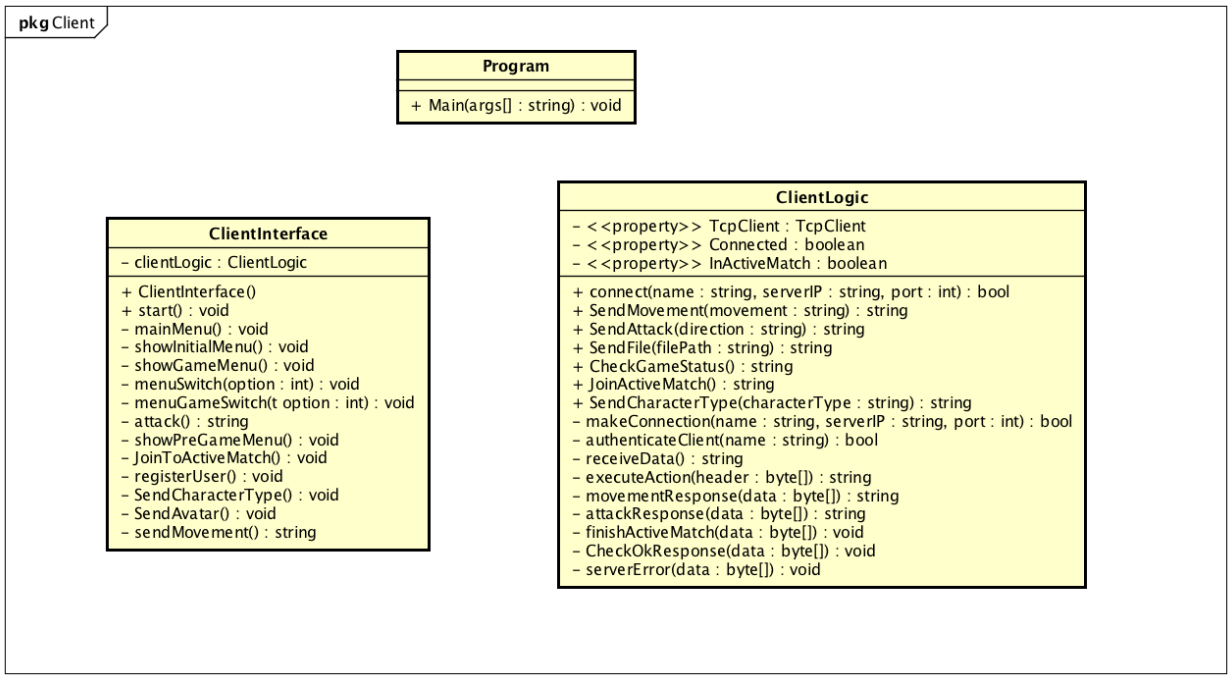


Diagrama del paquete Protocols

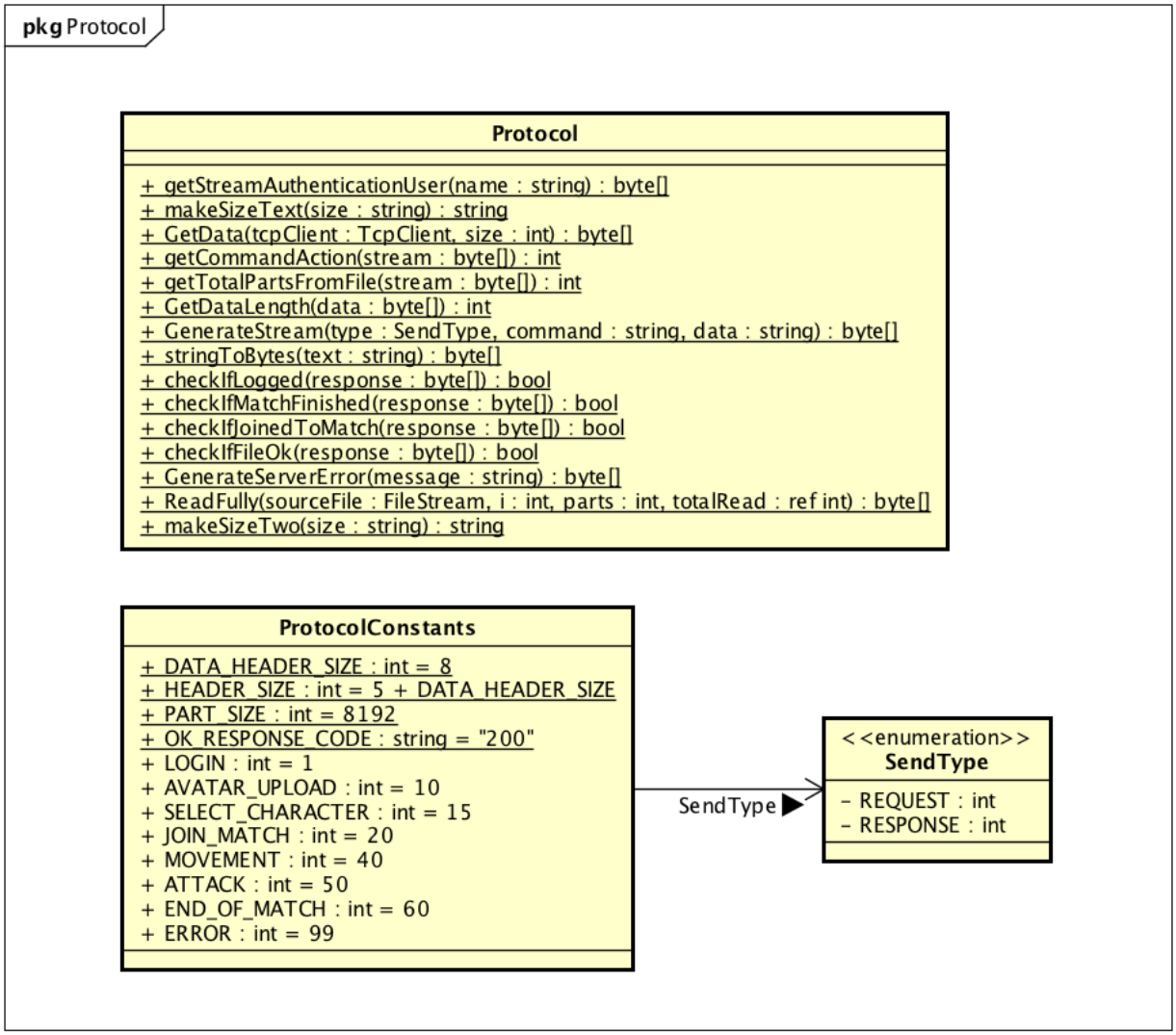
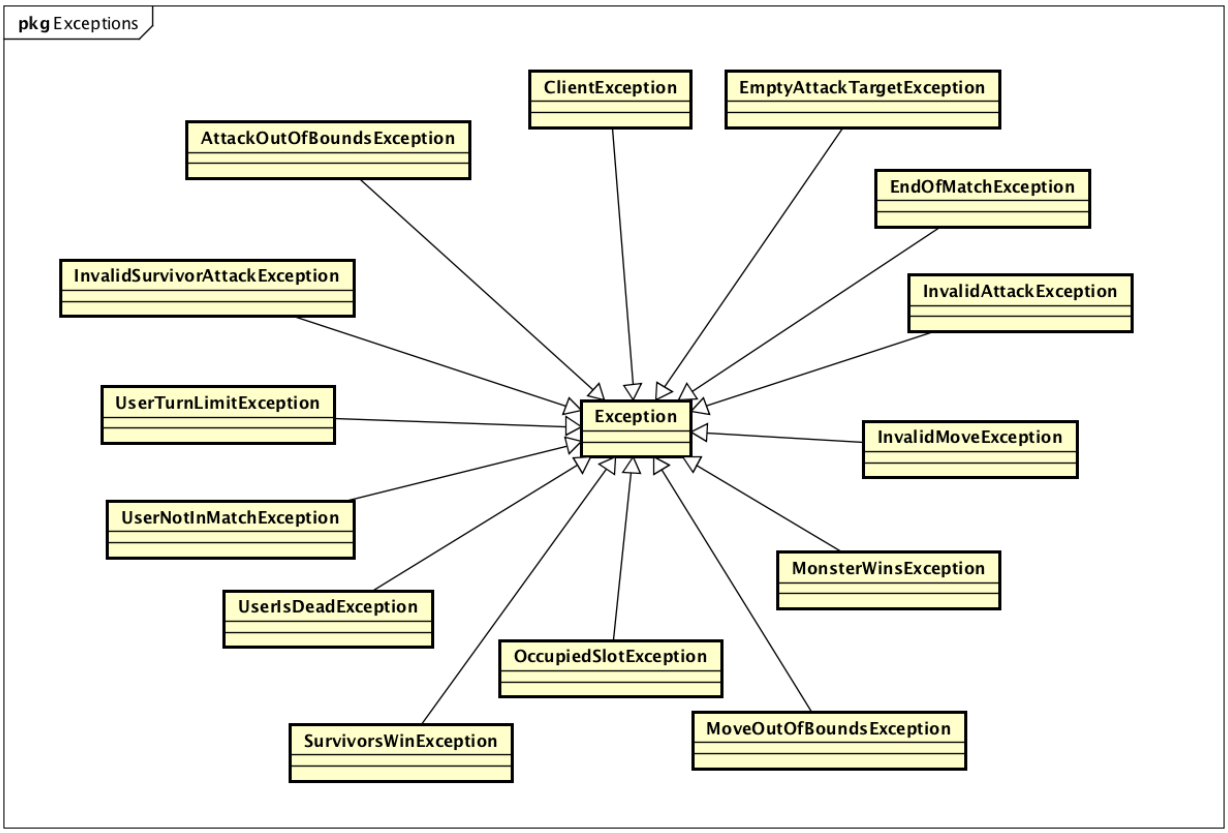
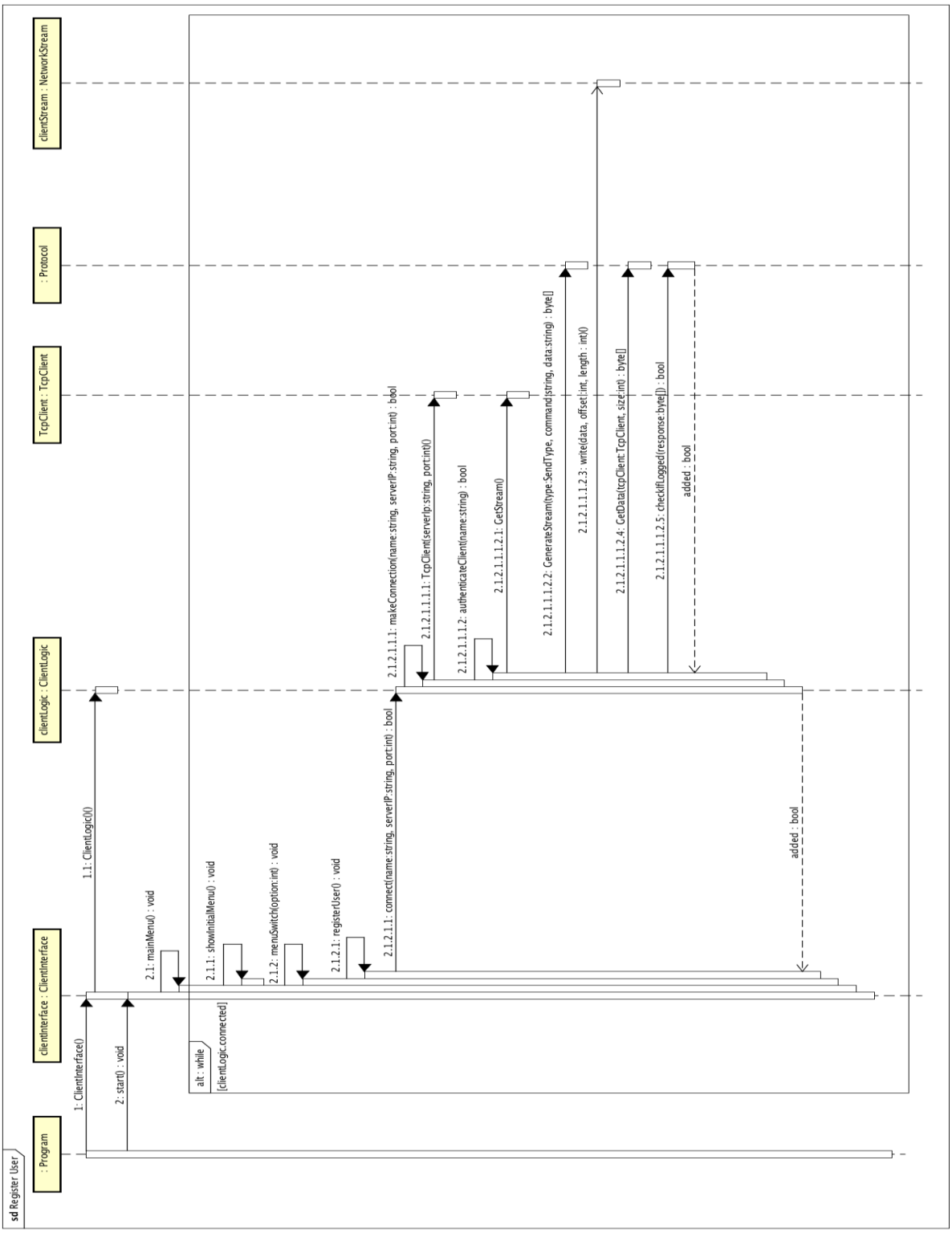


Diagrama del paquete Exceptions

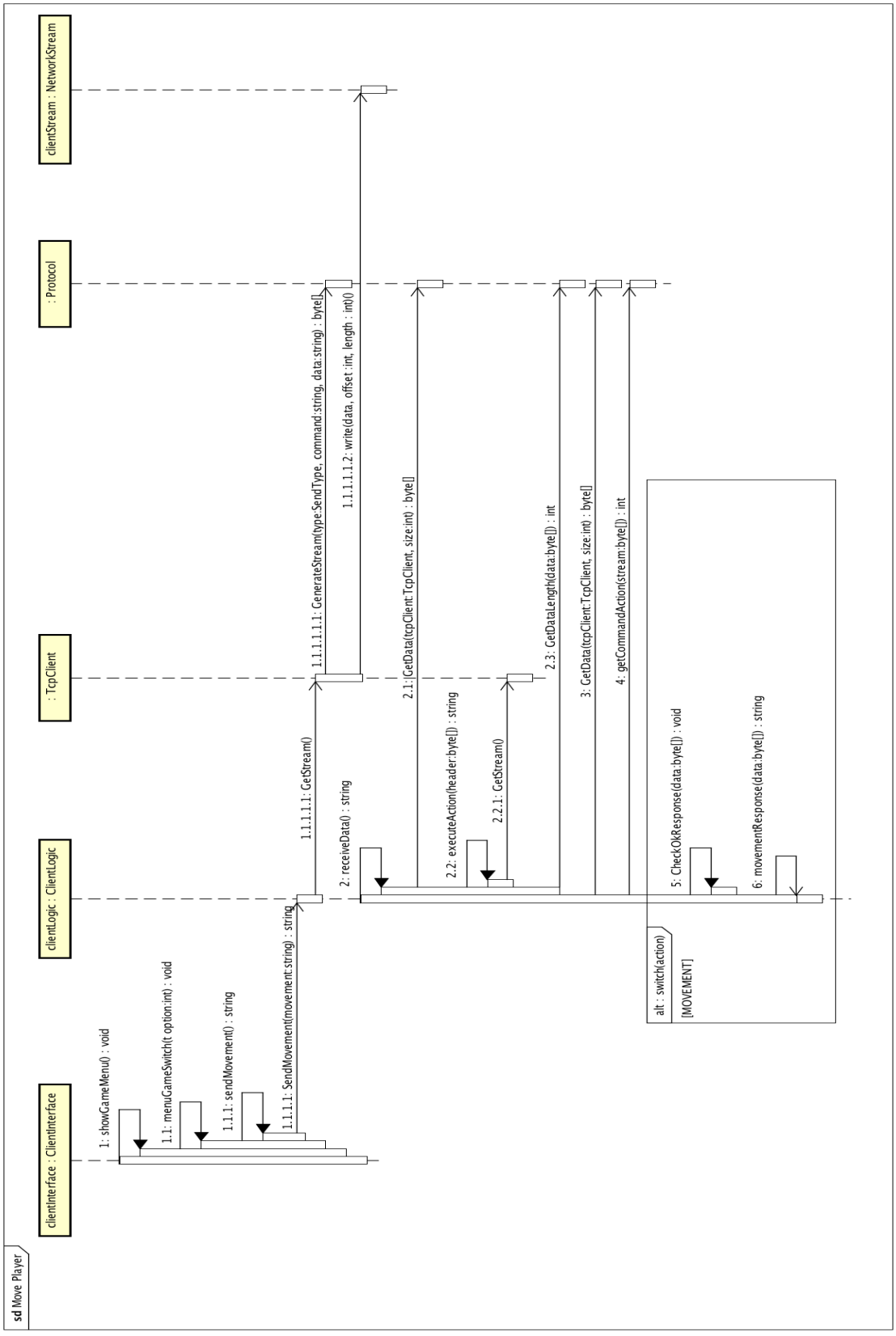


Diagramas de Secuencia

Registro de Usuario



Movimiento (análogo con ataque)



Subir archivo