MasterMind

Memoria del Proyecto

Redes y Videojuegos en Red
|
Grado en Desarrollo de Videojuegos
|
Universidad Complutense de Madrid

Por:
Sergio Alberto Luis Cano
Marcos Docampo Prieto-Puga

Proyecto en:
https://github.com/serluis/Redes_GDV

1. Descripción

a. ¿Qué es MasterMind?

El juego original MasterMind es un juego de razonamiento, con una edad +8, de menos de 20 min. de duración, y para dos jugadores. El objetivo del juego es adivinar una combinación de colores secreta que plantea el contrario. Esto significa que en cada ronda hay un jugador que codifica un patrón de colores y hay otro que debe adivinarlo, y si lo hace antes de que acaben todos los turnos, es el ganador.

b. ¿Cómo se juega a MasterMind?

En el MasterMind original en cada partida hay un atacante y un defensor. El jugador defensor propone un código de colores de 5 chinchetas al principio del juego. El defensor puede elegir entre 8 colores para crear la clave, y meter combinaciones a su antojo, pudiendo haber repeticiones de elementos sin importar el orden. Los 8 colores originales son: rojo, naranja, amarillo, verde, azul, rosa, blanco y gris.

Una vez el jugador ha creado la clave, el atacante dispone de 12 turnos para adivinar la contraseña. Todos los turnos tienen la misma mecánica: el atacante coloca 5 fichas de los colores que crea oportunos en las casillas designadas, y después el defensor usa las fichas blancas y negras para marcar los aciertos y errores del atacante en los marcadores de al lado de la propuesta.

¿Cómo funcionan las fichas blancas y negras del defensor? Muy fácil, el defensor colocará una ficha negra si el atacante acertó el color pero no el lugar, y una ficha blanca si acertó el color y el lugar de la chincheta. Por otra parte, si el atacante no acertó ninguna de las dos, el hueco marcador de acierto quedará vacío.

El juego contiene un tablero, que es más largo que ancho, teniendo una fila por cada turno, o sea, 12 filas, más otra fila, oculta, que es donde está la clave que el atacante debe sacar. Cada fila del turno se compone de cuatro agujeros grandes y cuatro pequeños, y un número que representa el turno en el que estamos, de la siguiente forma:

$$N^{o}\ Turno \quad Color\ 1 - Col\ 2 - Col\ 3 - Col\ 4 \quad Acierto\ 1 - Aci\ 2 - Aci\ 3 - Aci\ 4$$

Cada “Color” es un intento por resolver la posición en la combinación original, y cada “Acierto” es la respuesta a si se ha conseguido sacar algo de esa combinación. De esta forma, si el código propuesto es:

$$Rojo - Amarillo - V\,erde - Azul$$

Y el jugador propone en el turno 1:

$$1 \quad Rojo - V\,erde - Rosa - Naranja$$

Los marcadores de acierto serán:

$$Negro - Blanco - Nada - Nada$$

De tal forma que la línea del turno quedaría:

$$1 \quad Rojo - V\,erde - Rosa - Naranja \quad Negro - Blanco - Nada - Nada$$

c. Nuestra propuesta de juego

Nuestra propuesta del juego será una variante del juego original en la que habrá dos jugadores, que competirán por turnos para adivinar el código, creado por la máquina y defensora. El juego consta de una partida para los dos clientes atacantes que se irán turnando para introducir una propuesta de contraseña. El que primero adivine la contraseña se alzará con la victoria.

A diferencia del juego original nosotros dispondremos de 9 colores para dificultar un poco más el juego, y 12 turnos, los que implica que cada jugador dispone de 6 turnos para hallar la solución. Nuestros 9 colores son los siguientes: rojo, marrón, azul, amarillo, verde, morado, naranja, rosa, blanco y negro.

La variante que vamos a crear tiene como principal atractivo el juego en red para dos jugadores por turnos (escalable al número de partidas que deseemos), lo que establece una diferencia fundamental con el original en que solo uno de los jugadores introducía la contraseña y el otro era quien debía adivinarla.

Otra diferencia es que el histórico de fichas en la partida puede ser vista por los dos participantes, lo que implica que cada jugador puede ver la jugada del adversario. Esto hace que se tenga muy en cuenta que ha introducido el otro jugador ya que dará pistas al siguiente en cuanto a ganar se trata.

2. Arquitectura del Juego

a. Modelo del Juego

Nuestro juego tiene un modelo *cliente-servidor* usando el protocolo **TCP/IP**. Usamos el protocolo **TCP** porque es un protocolo de transporte orientado a conexión, sin pérdida de datos. Esto es perfecto para un juego por turnos.

En nuestro caso, el servidor tendrá la capacidad de manejar clientes de dos en dos, hasta tantos clientes como creamos necesarios. Cada 2 clientes se creará una partida independiente de todas las demás, solo con el factor común del servidor, pues solo hay uno.

El juego ha sido dividido en clases, en la que cada una maneja diferentes ámbitos del juego. Para empezar, la clase ***MMServer*** se encarga de crear un servidor en la dirección aportada, recoger a los clientes y meterlos en un hilo con una nueva instancia del juego. Por otra parte, ***MMClient***, maneja un único cliente, pues el código de esta clase corre una vez por cada cliente que creemos.

Luego encontramos el script ***Game***, que se encarga de manejar todo el juego. Dentro del script Game, podemos encontrar tres clases, ***GameServer***, ***GameClient*** y ***Message***. Las dos primeras clases se encargará de manejar la lógica del juego. ***GameClient*** además de la lógica maneja el renderizado del cliente.

La clase ***Message*** es la que se encarga de crear y procesar todos los mensajes entrantes y salientes que se transmiten entre el cliente y el servidor. ***Message*** guarda una profunda relación con ***Socket*** y su clase padre, ***Serializable***.

Socket y ***Serializable*** contienen las estructuras necesarias para crear servidores, clientes, manejarlos y crear la comunicación entre ellos. En estas clases se apoya todo el proyecto.

Por último, para el manejo de la parte gráfica hemos usado XLDisplay, una clase que maneja **Xlib**, una biblioteca que reúne un conjunto de funciones y macros realizadas en C y utilizadas por un cliente para hacer de interfaz con el servidor gráfico. [1]

Entonces, los archivos usados en el proyecto son:

- Game.cc Game.h
- MMServer.cc MMServer.h
- MMClient.cc MMClient.h
- XLDisplay.cc XLDisplay.h
- Socket.cc Socket.h
- Serializable.h
- Makefile

[1] Definición oficial de XLib

Con sus inclusiones y dependencias internas respectivas:

- Game.h: XLDisplay.h Serializable.h Socket.h
- MMServer.h: Game.h Serializable.h Socket.h <thread>
- MMClient.h: Game.h Serializable.h Socket.h
- XLDisplay.h: <stdexcept> <X11/Xlib.h> <stdlib.h> <string>
  <stdexcept> <memory> <vector>
- Socket.h: Serializable.h <string.h> <sys/socket.h> <sys/types.h> <netdb.h>
  <iostream> <stdexcept> <ostream> <vector> <sys/stat.h>
  <fcntl.h> <unistd.h> <fstream> <string>
- Serializable.h: <stdlib.h>

b. Estado del juego, Objetos y Replicación

En el juego existen tres posibilidades de estados. Hay que destacar que no existen los estados como tal en una máquina de estados, como podrían ser: menú, juego, créditos, niveles… Si no que tratamos estados como el momento en el que nos encontramos del juego, que solo se compone de la pantalla del MasterMind.

Entonces, en el juego existen las siguientes posibilidades: que estemos en partida, que haya ganado uno u otro jugador, o que ninguno haya ganado en los 12 turnos. El control de en qué momento estamos lo lleva el servidor, que se encarga de pasar de ronda, comparar datos para saber si alguien ha ganado y terminar en caso de que no haya nada más que hacer.

El paso de información se realiza mediante mensajes. En estos mensajes el primer dato es un entero cuyo valor fluctúa entre 0 y 2. En una ronda normal del juego, ese dato significa lo siguiente:

- 0: El juego continúa
- 1: Ha habido un ganador
- 2: El juego ha terminado sin ganador

No hace falta aclarar quien es el ganador porque solo puede haber ganado el jugador que está en su turno. Entonces, solo manejamos el primer dato en el turno del jugador que juega.

Sin embargo, cuando se conectan por primera vez los dos jugadores, el primer dato se procesa de otra manera: es usado para identificar quién juega primero. Como en cada partida solo puede haber dos jugadores, y estos son emparejados por orden de entrada, con cada conexión se envía a mano un 1 al jugador que ha entrado el primero, y un 0 al jugador que ha entrado el segundo. Cuando el cliente lo reciba, lo procesa asignándoselo a un booleano que les dice si es su turno.

c. Protocolo de Aplicación y Serialización

Dado que es un juego por turnos, en la que en cada turno solo hay que enviar los datos de las posiciones colocadas por el jugador al que le toca jugar, hemos optado por un tipo de mensaje simple. Según el planteamiento que hicimos, cada turno se compone de jugadas con 8 chinchetas, 4 del jugador del turno, y otras 4 de respuesta del servidor. Como cada chincheta está representada por un color que es un entero, para cada mensaje necesitaremos mínimo 8 enteros.

Por otra parte, como toda la lógica y comprobaciones de ganadores y perdedores se realizan en el servidor, también necesitaremos un flag del estado del juego, que le transmita la información de esto. Para esto usamos otro entero, por lo que en total tendremos mensajes de 9 enteros.

Estos mensajes de 9 enteros tienen el siguiente formato: primero el flag del estado del juego, después 4 enteros más que representan los colores del intento, seguidos de los 4 colores de acierto y fallo:

$$Estado - Col\ 1 - Col\ 2 - Col\ 3 - Col\ 4 - Aci\ 1 - Aci\ 2 - Aci\ 3 - Aci\ 4$$

En el que las parámetros significan:

	Estado	Col	Aci
Significado	El estado del juego, o sea, si alguien ha ganado, perdido o ninguno de los dos ha ganado.	El color con el que el jugador trata de adivinar la solución.	Respuesta del servidor sobre si es correcto el color *Col*.
Rango de ***int***	0 - 2	0 - 9	8 - 9

En la que el orden de colores sigue el siguiente esquema:

int	Color	Representacion
0	ROJO	
1	MARRON	
2	AZUL	
3	AMARILLO	
4	VERDE	
5	MORADO	
6	NARANJA	
7	FUCSIA	
8	BLANCO	
9	NEGRO	

Existen dos colores más, el siena y el perú, dos tonos suaves de marrón con los identificadores 10 y 11 que se usan para el tablero. El 10 es el color de fondo y el 11 es el color de los huecos donde van las chinchetas. Estos colores también son enviados en los casos en los que la línea aún no está completa.

int	Color	Representacion
10	SIENNA	
11	PERU	

La serialización es un proceso bastante simple en nuestro caso. Dado que solo enviamos enteros y existe una clara ausencia de variedad en tipos, optamos por **serializar** *todos los números uno a uno*, sin necesidad de interacción con un array o un vector. Hay que aclarar que solo en la creación de mensajes y su manejo en la serialización no son usadas estructuras de datos, puesto que al crearse la contraseña y las comparaciones en el servidor si que se realizan con vectores.

d. Línea de ejecución y controles

Para ejecutar el proyecto es indispensable estar en un SO Linux. Debemos bajarnos el último commit del repositorio del proyecto, y meterlo en una única carpeta. Desde la consola, y encontrándonos en ese directorio, ejecutamos:

```
$ make
```

Cuando ya se haya compilado el proyecto, se habrán creado dos archivos ejecutables, ***server*** y ***client***. Estos dos archivos requieren dos argumentos cuando se vayan a ejecutar: la **IP** y el **puerto** del servidor, en ese estricto orden.

Entonces, en una consola ejecutamos el servidor, y en otras dos diferentes ejecutamos el cliente, de la siguiente forma:

```
$ ./server 127.0.0.1 7777
$ ./client 127.0.0.1 7777
```

Después de esto se abrirán dos ventanas, cada uno siendo un jugador del MasterMind. Jugará primero el jugador que se conectó antes, teniendo por defecto cada casilla de colores en rojo. Tendrá las siguientes opciones con los siguientes controles.

Tecla	Acción
‘w’	Cambiar casilla al siguiente color
‘s’	Cambiar casilla al anterior color
‘a’	Cambiar a la casilla de la izquierda
‘d’	Cambiar a la casilla de la derecha
‘e’	Enviar la combinación de colores

Además de una tecla adicional que solo se puede pulsar cuando termina la partida:

Tecla	Acción
‘q’	Salir del juego

3. Diseño del Servidor

a. Resumen del Servidor

El objetivo del servidor es fundamentalmente recibir los *Socket* de los clientes, establecer la comunicación entre ellos y sí mismo y mandarlos a **Game** para iniciar la partida, verificar que no haya fallos en las conexiones de los clientes y cerrar adecuadamente el juego cuando hay algún fallo en la conexión inicial o cuando la partida acaba.

b. Diseño interno

i. Private

Contiene una declaración de *Socket*:

```
Socket socket;
```

ii. Public

Contiene una constructora **MMServer(...)** que tiene por parámetros dos punteros constante a *char*, una asignación de *socket* con una IP y un puerto por parámetros y en su interior se llama a la función **bind(...)** de socket.

```
MMServer(const char *ip, const char *port) : socket(ip, port) {
	socket.bind();
};
```

Tiene también una destructora vacía y un *getter* que devuelve un Socket, llamada **getSocket(...)** sin parámetros.

```
~MMServer() {};
Socket getSocket() { return socket; };
```

Nuestro *MMServer.cc* tiene una función **main(...)** con un *int* y un *char*** por parámetros, que serán la dirección IP de la máquina que lo lanza y el puerto en el que se va a alojar.

```
int main(int argc, char** argv)
```

Se llama a la función constructora de la clase con los anteriores parámetros.

```
MMServer server(argv[1], argv[2]);
```

Se realiza un *cout* por consola para informar de que se está ejecutando correctamente.

```
std::cout << "== Servidor alojado en: " << server.getSocket()
<< " ==\n" << std::endl;
```

Se crea un *vector* de ***threads*** llamado pool, se genera una constante int ***MAXPARTIDAS*** la cual está actualmente limitada a dos, pero puede ser un nº mayor. Luego se crea un *int* partidas que irá contando cuantas partidas se inicializan para no superar el límite.

```
std::vector<std::thread> pool;
const int MAXPARTIDAS = 2;
int partidas = 0;
```

Se inicia un *while* que tiene por condición que partidas sea inferior a ***MAXPARTIDAS***. Dentro de éste lo primero que se hace es incrementar las partidas.

```
while (partidas < MAXPARTIDAS) {
	++partidas;
```

A continuación se realizan dos *accept* con dos cout para informar de que *sockets* tienen.

```
Socket* sock_client_one = server.getSocket().accept();
std::cout << "== Cliente 1 conectado desde: " << *sock_client_one
<< " ==" << std::endl;
Socket* sock_client_two = server.getSocket().accept();
std::cout << "== Cliente 2 conectado desde: " << *sock_client_two
<< " ==" << std::endl;
```

Después se crea un condicionante de seguridad para asegurar que ninguno de los socket está vacío. Se crea un *thread* y se mete en el vector con las siguientes órdenes:

```
pool.push_back(std::thread([&]() {...}
```

Se llama a la constructora de *Game.h*, con los socket del servidor y de los clientes como parámetros.

```
GameServer MasterMind(server.getSocket(), sock_client_one,
sock_client_two);
```

Después se inicia otro *while* con la función ***getEnd(...)*** de MasterMind por condición y se llama a la función ***update(...)*** de **MasterMind** dentro del bucle. Luego se crea otro *cout* de fin del juego para cuando sale de él.

```
while(!MasterMind.getEnd())
	MasterMind.update();
std::cout << "== Fin del juego ==" << std::endl;
```

Posteriormente existe un *else* como alternativa al *if* anterior de comprobante de conexión de los *socket* de los clientes en el que se genera otro *cout*. Otro *cout* más informa de cuántas partidas están funcionando hasta ese momento.

Por fin para finalizar se crea un for que hará *join* de todas las partidas que terminan, informando por medio de *cout* de que se ha terminado la conexión con el servidor y este se ha cerrado.

```
	for (auto &t : pool) {
		t.join();
		std::cout << "== Conexion terminada ==" << std::endl;
	}
	std::cout << "== Servidor cerrado ==" << std::endl;
	return 0;
```

4. Diseño del Cliente

a. Resumen del cliente

La función de los clientes es enviar su *socket* y su IP al servidor para establecer la comunicación y llamar a la función de Game que inicia el juego.

b. Diseño interno

i. Private

En *private* se crea un *Socket* sin inicializar.

ii. Public

En la parte *public* está la constructora, la destructora vacía y un *getter*. La constructora tiene como parámetros dos punteros a *char* constantes y una asignación server con esos dos parámetros. El getter devuelve un *Socket* llamado server.

```
MMClient(const char *ip, const char *port) : server(ip, port)
```

Nuestro *MMClient.cc* tiene un main con un *int* y un *char*** por argumentos que servirán para llamar a la constructora del cliente.

```
MMClient client(argv[1], argv[2]);
```

Después se crea un entero llamado connection y se inicializa con una función del *socket* que servirá para conectarlo al servidor. El entero se usará posteriormente en un *if* condicional para comprobar la correcta conexión del cliente.

```
int connection = client.getSocket().connect();
if (connection != -1) { … }
```

Después se hace un *cout* para informar de la conexión del cliente si ha sido exitosa, o en caso contrario entrará en un *else* que informará acerca del fallo de conexión.

```
std::cout << "== Cliente conectado ==" << std::endl;
else std::cout << "== Error de conexion ==" << std::endl;
```

Luego se llama a la constructora de la clase **GameClient**, que recibe como parámetros el *socket* del cliente y dos enteros que representan el tamaño de la ventana de juego que se abrirá a continuación.

```
GameClient MasterMind(client.getSocket(), 400, 450);
```

Por último se inicia un *while* que tiene por condición un *bool* que proporciona un *getter* de la clase **GameClient** y que tiene dentro una función de la misma clase anterior que inicializa el flujo de juego en sí.

```
	while(!MasterMind.getEnd())
		MasterMind.update();
```

5. Librerías y Dependencias

Lenguaje: C++
Sistema operativo: Linux Redhat/Ubuntu (usado en máquinas virtuales).
Librerías gráficas: <X11/XLib.h>

6. Seguridad y Errores

a. Seguridad

Existen mensajes de control para todas las interacciones de tipo red. Por ejemplo en el archivo MMServer.cc:

```
if (sock_client_one != nullptr && sock_client_two != nullptr) {
	pool.push_back(std::thread([&]() {
		std::cout << "== Creado thread con: " << *sock_client_one
		<< " y " <<  *sock_client_two << " ==" << std::endl;

		GameServer MasterMind(server.getSocket(), sock_client_one,
			sock_client_two);
		while(!MasterMind.getEnd())
			MasterMind.update();
		std::cout << "== Fin del juego ==" << std::endl;
	}
}
else std::cout <<
  "== Los clientes no se han conectado correctamente ==" << std::endl;
```

```
if (connection != -1) {
	std::cout << "== Cliente conectado ==" << std::endl;
}
else std::cout << "== Error de conexion ==" << std::endl;
```

Si esto sucede, es inevitable tener que volver a lanzar el cliente para intentar reconectar. Quizás se podría hacer algún sistema que lo reinicie solo hasta obtener una respuesta satisfactoria o tras varios intentos infructuosos informar de que la conexión ha sido imposible de realizar definitivamente.

b. Errores

Es importante destacar que se puede manejar con las teclas de controles al jugador que no está jugando. Esto creará un conflicto en el caso de que se envíe una combinación antes de que sea su turno, y congelaría el juego, sin más solución que abortar el código y volverlo a correr.

7. Conclusiones

Los aspectos más importantes del juego fueron la *planificación inicial*, en la que debíamos elegir cuál iba a ser el juego a elegir y cómo íbamos a planear el acercamiento a este, los problemas que derivarían al irse creando las clases y cómo se iban a comunicar entre ellas.

Es cierto que no se puede planear todo de antemano y que los problemas surgen inevitablemente, pero también refleja un buen trabajo el ser capaz de adaptarse a la situación y de prever los máximos problemas que puedan surgir para estar preparado para afrontarlos.

Una vez establecido que una variación del juego MasterMind era viable, el cómo establecer el tipo de conexión era lo siguiente. Decidimos usar una conexión por **TCP** y adaptar las clases que ya teníamos creadas anteriormente para ahorrar tiempo. El juego iba a ser por turnos así que debíamos decidir también que mensajes se iban a enviar y cual sería su formato. Un formato de 9 enteros fue la solución. un primero para decidir si la partida había terminado, que también se podía utilizar al principio en un primer mensaje para establecer el turno de los jugadores. Aquí como posible mejora se podría haber añadido un generador de números aleatorio para que el turno no siempre correspondiera al jugador que primero se conecte.

Los cuatro enteros siguientes serían destinados a la contraseña de color del jugador correspondiente y los últimos cuatro para la respuesta del servidor con las pistas acerca de la contraseña.

Un dato a destacar fue que tuvimos que adaptarnos al uso de la librería gráfica de **Xlib/X11** una librería del todo desconocida para nosotros hasta el momento. Fue muy fácil de usar y se adaptó a nuestras necesidades. Dado parte del código de ésta por el profesor y con unas pequeñas adaptaciones hicimos el trabajo con suficiencia.

Tuvimos ciertos problemas con el paso de mensajes, no así con la serialización como tal y su deserialización. La mayor parte del problema estuvo en la estructura de los mensajes. Dado que solo usamos 9 ***int***, queríamos aprovechar al máximo la optimización de mensajes y usar ***int8_t***, pero esto nos generó errores a la hora de configurar los mensajes, así que lo dejamos como ***int***

Después de acabar la fase principal en la que queríamos tener el juego funcional, decidimos añadir una manera de que pudieran jugar más de dos jugadores. Establecimos un sistema en el cual se crearían threads con cada dos jugadores para cada partida y un límite de partidas para que pudiera regularse cuantas queríamos en cada momento. Se quedó establecido en dos pero se puede ampliar hasta los límites del paso de mensaje y del servidor o del puerto.

Otro de los problemas fue que los jugadores pese a no estar en su turno podían manejar la siguiente contraseña sin verla al presionar las teclas establecidas para el control del input. (Aún no está solucionado). Esta sería una de las mejoras a tener en cuenta y que deberían solucionarse lo antes posible.

8. Bibliografía y referencias

- *https://mathworld.wolfram.com/Mastermind.html -Breve descripción.*