Integrantes:		*Nro. Alumno:*	*Grupo:*
Zingarelli Facundo	-	66092/1	5
Sarin Gaston	-	62009/1	
Amantea Tomas	-	65520/5	

FACULTAD DE INGENIERÍA

Informe de programa

TP Integrador

ENUNCIADO:

"Se desea realizar un servidor TCP que permita jugar una partida del tradicional juego de cartas "Escoba de 15", desde un cliente telnet El servidor debe esperar conexiones entrantes en el port 1234 y deberá incorporar, como mínimo, las siguientes capacidades:

A. *Permitir jugar una partida entre 2, 3 o 4 jugadores.*
B. *Cuando se conecte el primer jugador, se deberá ofrecer la posibilidad de seleccionar si la partida aceptará 2, 3 o 4 jugadores.*
C. *Una vez seleccionada la cantidad de jugadores, deberá crear e inicializar los recursos necesarios y esperar que se presenten el resto de los jugadores.*
D. *Una vez que se hayan conectado el resto de los jugadores, se repartirán las cartas y se avisará al primer jugador conectado que es el que inicia la partida ("mano"), el orden de participación del resto de los jugadores deberá ser el mismo que el orden de conexión.*
E. *El servidor enviará a cada jugador conectado la información sobre cuales naipes le tocaron en el reparto y además cuales son las naipes que están sobre la mesa.*
F. *El servidor habilitará al jugador que tiene el turno de juego a enviar su jugada, una vez recibida la reenviará a todos los jugadores.*
G. *Luego informará a cada jugador cuáles son los naipes que tiene en su poder, cuáles son los naipes que quedan en la mesa y quien tiene el próximo turno.*
H. *Si un jugador intenta enviar su jugada cuando no le toque el turno, el servidor ignorará el intento.*
I. *El juego finaliza cuando no hay más cartas para repartir.*
J. *Cuando finaliza el juego, el servidor informa a todos los jugadores, las cartas recolectadas por cada jugador y la cantidad de escobas para poder realizar un recuento manual de puntaje."*

INTERPRETACIÓN:

- Se desea realizar un juego de cartas en un servidor TCP
- El servidor esperará conexiones entrantes por puerto 1234.
- La implementación incluye:
    - Partida de 2-4 jugadores. El primero selecciona la cantidad de jugadores que participarán.
    - Un mensaje de confirmación de conexión entre cliente-servidor previo al pasaje de archivos / datos.
- Luego quedará a la espera de que se conecten todos los participantes.
- Una vez conectados, se repartirán las cartas y se le avisará al primero en conectarse que es quien inicia la partida, el resto de jugadores deberán esperar a su turno, el cual será designado en orden de conexión(es decir jugará antes quien haya entrado antes).
- El servidor informará a todos los jugadores las jugadas que se vayan realizando.
- El jugador solo podrá jugar en su turno, ya que el servidor le impedirá realizar cualquier maniobra en otro momento.
- Una vez se hayan repartido todas las cartas(y luego de haberse jugado todas las rondas) finalizará el juego, en ese momento el servidor informa a todos los jugadores las cartas recolectadas por cada jugador y la cantidad de escobas para luego poder realizar un recuento manual del puntaje.
- El server implementado carece de una interfaz gráfica dedicada, toda la información que se muestra al/los jugador/es se hace mediante mensajes por socket, y en formato texto. Por una cuestión meramente estética se utilizaron símbolos : guiones, paréntesis, números,etc.
- La lógica a través de la cual el jugador indica qué o cómo jugará, se realiza con respuestas numéricas e índices asociados a las opciones de respuesta que tiene. Ej:

  "Ingrese (1) si desea jugar,o (2) si desea descartar: "

  o:

  "MANO: ---3 de Copa (1)--- ---7 de Espada (2)--- "

- Utilizamos terminología específica, para poder diferenciar precisamente a qué etapa del juego nos referimos:
    - ❖ JUGADA: Corresponde al descarte o levante de UN jugador. Ej: Pepe jugó el 7 de Oro y levantó el 10 de Espada
    - ❖ MANO: Corresponde a una jugada por jugador. Ej: Para 3 jugadores una mano corresponde a 3 jugadas.

- ❖ RONDA: Corresponde a 3 manos.

- Bajo estos términos, una partida consta siempre de un número entero de RONDAS, ya que de las 40 cartas se sitúan 4 sobre la mesa la 1era ronda, quedando 36 a distribuirse entre 2, 3 o 4 jugadores. Y por ende, una partida consistirá siempre de n.3(manos), con n entero.

RESOLUCIÓN:

Pseudocódigo:

Servidor

```
INICIO

Definición de puertos, protocolos e IP;
Definición de variables globales,estructuras,etc.
Declaracion función iniciar mazo;
Declaracion función mezclar mazo;
Declaracion función de primera mano;
Declaracion función cambio de carta(12,11,10);
Declaracion función levantar cartas;
Declaracion función descartar;
Declaracion función carga de la mano;
Declaracion función carga de la mesa;
Declaracion función cargar cartas levantadas;
Definición de variables locales;
Inicialización cola de mensajes;
Inicialización memoria compartida;
Creacion de socket;
Seteo de datos del server;
while(!terminar){
```

```
Espera bloqueante de conexiones entrantes;
Manejo de errores;
if(jugador1 ){
  Mensaje de bienvenida;
  Escaneo de cantidad de jugadores que jugaran;
  }
  Se muestra info del proc padre y cliente;
       }
   Creación de hijo{
        close (server_s);
        impresión de la información del cliente y del hijo;
        Seteo de timeout para socket cliente;
        Mensaje de confirmación al cliente;
        If(resto de jugadores){
        Mensaje de bienvenida;
        Manejo de errores;
        Seteo de variables del jugador y mensajes al mismo;
        while(ronda){
         Se reparten las cartas;
         Mensaje de confirmación (a proceso padre);
         Mensaje de confirmación(al cliente)si quedan rondas;
         Loop de ronda(tres manos){
          Carga la mano y se la envía al cliente;
          Espera que el padre habilite acceso a memoria
compartida;
          Carga de la mesa;
          Confirmación de acceso a M.Compartida;
          Mensaje de aviso al jugador en espera;
          Jugador recibe las jugadas de jug PREVIOS a sí;
          Aviso de jugador a jugar;
          do{
              Carga de  la mano y se la envía al cliente;
              Carga la mesa y se envía al jugador;
              Loop hasta que se ingrese una JUGADA VÁLIDA;
```

```
                jugada = true;
                while(jugada){
                  Mensaje de aviso al jugador;
                   Recibe la decisión del juagor y la almacena;
                   If( levantar){
                      Mensaje de aviso al jugador;
                      Recibe carta a jugar y la almacena;
                      Mensaje de aviso al jugador;
                      Recepción y guardado de número de cartas a
levantar;
                      Recepción y guardado de las cartas a
levantar;
                      Manejo de errores;
                      Evaluación de suma correcta;
                      Evaluación de escoba;
                      Se avisa al proc padre que el jugador
terminó de jugar;
                      terminar2 = true;
                      jugada=false;
                      }
                   else if( descartar ){
                      mensaje de aviso al jugador;
                      Se recibe la carta y se almacena;
                      Se llama a función descartar;
                      Se avisa al proc padre que el jugador
terminó de jugar;
                      terminar2 = true;
                      jugada=false;
                      }while(!terminar2);
                Se reinicia var de control;
                Jugador recibe las jugadas de los jug POST;
```

```
                Recepción msje de proxima mano;
                Se avisa al jugador que no se jugarán más rondas;
                Se cargan cartas recolectadas;
                Termina el hijo;
                close(connect_s);
                }TERMINA EL HIJO
        Continúa el padre:
        Espera del total de jugadores;
        Llamado a funcion inicializar mazo;
        Llamado a función mezclar mazo;
        Llamado a función primera mano;
                loop juego:
                while(Ronda){
                        Se reparten las cartas a cada jug;
                        Espera de confirmación;
                        Habilita a cada jugador a acceder a la MESA;
                        Espera confirmación del jugador;
                        }
                        Loop de manos(3){
                        Habilita a cada jugador a acceder a la MESA;
                        Recibe la jugada del jugador;
                        Manejo de errores;
                        Indica quién será el próximo en jugar;
                        Envía la jugada al resto de jugadores;
                        Evalúa si se dieron todas las cartas del
mazo;
                        En caso de ser así avisa a todos los jug que
no se jugarán más rondas;
                        }
        }
```

```
            close(connect);
            close(server_s);
      }
FIN
```

Detalles de implementación:

La estructura general del servidor se ilustra en la sig. imagen:

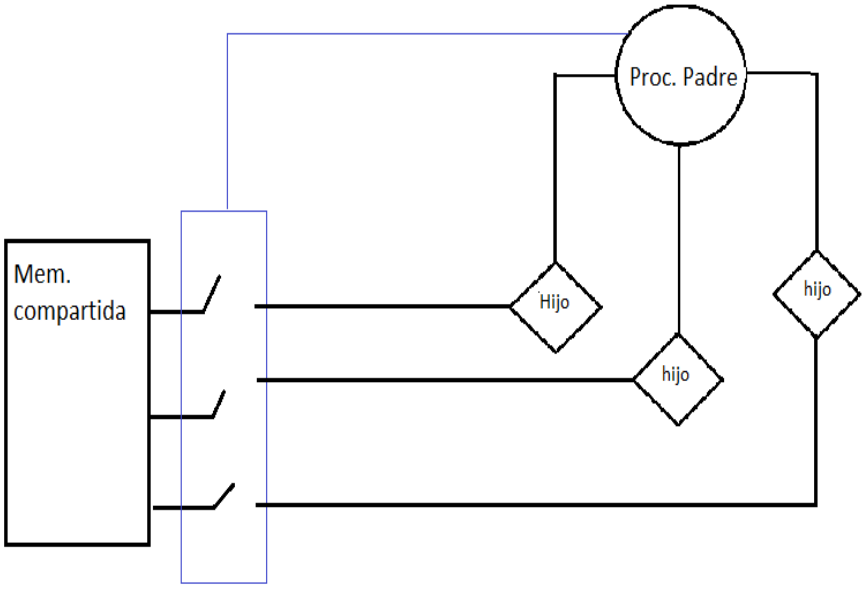

Imagen 1.0 :Estructura general del servidor

Como se puede apreciar, es una estructura jerárquica, en la que el proceso padre se encarga no sólo de llevar: La repartición de cartas, sino que además sincroniza el acceso a memoria compartida para que no haya data races ni se corrompan los datos almacenados.

Bajo esta idea, existirá un hijo por cliente, y cada uno estará encargado de:

- Mantener la conexión (socket) con el cliente, y enviarle mensajes, así como recibir y procesar su respuesta.
- Una vez realizada una jugada, cargarla como texto en un mensaje (Cola de msjes) y enviársela al proceso padre.
- Evaluar que las jugadas sean correctas, y acceder a memoria compartida para modificar las cartas restantes en mesa.

- Cargar en un struct (Jugador) las cartas que recolectó, la mano que tiene/le queda, y la cantidad de escobas que va realizando.

Por otro lado el proceso padre hará de “director de orquesta”, y se encargará principalmente de la sincronización. Sus tareas serán:

- Asegurar que la partida comience sólo cuando estén todos los jugadores conectados.
- Inicializar el mazo, así como encargarse de repartir las cartas cada ronda.
- Sincronizar, mediante cola de mensajes, el orden de los turnos, el acceso a memoria compartida, etc.
- Recibir las jugadas de los jugadores, y re-dirigirla hacia el resto de los jugadores, por cola de mensajes.

Dicho esto, podemos ahora incursionar en lo que sería detalles de cómo esta estructura fue implementada.

Variables globales(Defines):

En la sección de DEFINES encontramos secciones referidas al server, a memoria compartida, y al juego. Muchas han ya sido utilizadas previamente así que describiré las pertenecientes al juego:

```
//JUEGO
#define CARTAS_MAZO 40
#define CARTAS_MANO 3
#define NRO_MAX_CARTAS_MESA 20
#define NRO_MAX_CARTAS_REC 40


#define PASADAS_RAND 100
#define UPPER 39
#define LOWER 0
```

Se definieron de esta manera principalmente con finalidades de testing, ya que en realidad la cantidad MÁXIMA de cartas del mazo o la mano por ej es siempre fija.

Por otro lado, las 3 referidas al algoritmo pseudo-aleatorio que utilizamos, sirven para:

- `#define PASADAS_RAND 100` -> Determina la cantidad de loopeos utilizados a la hora de "mezclar" el mazo, que será descrita posteriormente.
- `#define UPPER 39` && `#define LOWER 0` -> Se utilizan para determinar los límites, o rango de números, que devolverá el algoritmo pseudo-aleatorio. En nuestro caso buscamos que devuelva números enteros entre 1 y 40.

Structs:

Uno de los principales énfasis en este diseño fueron los structs, ya que había muchas estructuras de datos que tenían atributos en común o podían agruparse para modularizar más el código. Empezamos por el más utilizado y que es el núcleo del juego:

```
struct Carta{
      char palo[6];
      unsigned int val;
};
```

De esta forma, cada carta queda caracterizada por un String, y un número, representando su valor y su palo: "3 de Copa".

Para ello definimos globalmente un array de Strings constante que contiene:

```
char *palos[4] = {"Basto" , "Espada" , "Copa" , "Oro"};
```

De esta forma al crear cartas simplemente debemos inicializar 2 atributos de un struct, uno con su valor y otro asignando alguno de los palos anteriormente definidos.

Seguimos con el mazo:

```
struct Carta mazo[CARTAS_MAZO];
unsigned int indice_mazo = 0;
```

Como se puede apreciar, el mazo no es más que un array de tamaño fijo, conformado por elementos de tipo "Carta". Y la variable GLOBAL indice_mazo, sirve para llevar cuenta de qué cartas se repartieron. Se abordará más al respecto al describir las funciones referidas al mazo. Seguimos con el struct Jugador:

```
struct Jugador{
```

```
        struct Carta cartas_rec[NRO_MAX_CARTAS_REC];
        unsigned int nro_c_rec;
        char nombre[20];
        int escobas;
        struct Carta mano[3];
        unsigned int nro_c_mano;
};
```

Este struct será inicializado y actualizado/modificado en cada proceso hijo asociado a su respectivo cliente. De sus atributos podemos ver que hay dos arrays con elementos tipo “Carta”, que son cartas_rec, y mano, correspondientes a las cartas recolectadas y la mano del jugador, respectivamente. Además hay dos variables contadoras: nro_c_rec y nro_c_mano, que llevan cuenta de cuántas cartas se recolectaron, y cuántas cartas le quedan en la mano, respectivamente.

Por último tenemos la var contadora escobas, que será incrementada cuando el proceso hijo correspondiente al cliente que esté realizando una jugada detecte que al levantar cartas, no quede ninguna restante en la mesa. Y el array de tipo char, donde se almacenará el nombre de cada jugador.

Nos queda entonces:

```
struct mesa_compartida{
        struct Carta cartas_mesa[NRO_MAX_CARTAS_MESA];
        unsigned int nro_c_mesa;
        char nombres_jug[4][20];
        int ultimo_levante;
};
```

Este struct es el que conformará la memoria compartida entre procesos. Consta de las cartas correspondientes a la mesa, una variable contadora que lleva cuenta de cuántas quedan. Los nombres de los jugadores (Que son recibidos por cada hijo, pero el padre necesita poder acceder a ellos), y una var contadora que se actualiza luego de terminar cada jugada válida, que será un número correspondiente al número de jugador que realizó la jugada, de esta forma al terminar el juego, esta variable representará quién jugó último, y por ende, quién se lleva las cartas que puedan sobrar/quedar en la mesa.

Por último tenemos 3 structs correspondientes a los 3 tipos de mensajes distintos que utilizamos:

```
typedef struct{
	long id;
	bool flag;
}mensaje1;


typedef struct{
	long id;
	struct Carta carta;
}mensaje2;


typedef struct{
	long id;
	char jugada[200];
}mensaje3;
```

La finalidad de estos mensajes es:

- **Mensaje1** -> Se utilizan para sincronizar. Ya sea el acceso a memoria, o para que el proceso padre avise a los proc hijos que no se jugarán más rondas.
- **Mensaje2** -> Se utilizan para repartir las cartas del mazo a cada jugador.
- **Mensaje3** -> Se utilizan para almacenar (Como TEXTO), las jugadas de los jugadores, y poder comunicarlas entre hijo-padre-resto de los hijos.

Funciones:

- **Inicializar el mazo:**

  Esta función crea un mazo de 40 cartas, ordenado por número y palo a través de iteraciones

```
void inic_mazo(){
	unsigned int cont=0;
	int i,j;
	for(j=0 ; j<4 ; j++){
		for(i=1 ; i<=7 ; i++){
			strcpy(mazo[cont].palo ,palos[j]);
			mazo[cont].val = i;
			cont++;
		}
		for(i=10 ; i<=12 ; i++){
			strcpy(mazo[cont].palo ,palos[j]);
			mazo[cont].val = i;
			cont++;
		}
	}
}
```

- **Mezclar mazo:**

  Luego se realizan un número de intercambios finitos entre cartas cuyos índices se determinan aleatoriamente. Como el número de intercambios es alto respecto a la cantidad de cartas, se considera que el mazo estará completamente desordenado.

  Luego se reparten las cartas de forma secuencial con el mazo desordenado.

```
void mezclar_mazo(){
	srand(time(0));
	int i;
	for(i=0 ; i < PASADAS_RAND ; i++){
		int j = (rand() % (UPPER - LOWER + 1)) + LOWER;
		int k = (rand() % (UPPER - LOWER + 1)) + LOWER;
		struct Carta carta_aux;
		carta_aux = mazo[j];
		mazo[j] = mazo[k];
		mazo[k] = carta_aux;
	}
}
```

- **Primera mano:**

  Esta función se encarga de cargar en en la memoria compartida, las primeras 4 cartas correspondientes a la mesa.

```
void primera_mano( int nro_jug, struct mesa_compartida *mem_comp
){

unsigned int i;

mem_comp->nro_c_mesa = 4;

for(i=0 ; i< mem_comp->nro_c_mesa ; i++){

        mem_comp->cartas_mesa[i] = mazo[indice_mazo];

        printf("---%d de %s---
\n",mem_comp->cartas_mesa[i].val,mem_comp->cartas_mesa[i].palo);

        fflush(stdout);

        indice_mazo++;

}


}
```

- **Convertir valor:**

  Esta función toma como argumento una variable entera, y transforma el número de la carta, al valor utilizado en el juego para sumar 15:

```
int convert_valor(int a){
	switch(a){
		case 12:
			return 10;
		break;
	case 11:
		return 9;
		break;
	case 10:
		return 8;
		break;
	default:
		return a;
	}
}
```

- **Levantar y descartar cartas:**

Estas funciones son las encargadas de tomar como argumento las cartas jugadas/descartadas por el jugador, modificar adecuadamente las cartas restantes en mesa/mano, agregar las cartas levantadas al struct Jugador, etc.

```
bool levantar_cartas(struct Jugador *jug,struct mesa_compartida
*mem_comp,int c_jugo , int *c_lev, int nro_c_lev )

void descartar_carta(struct Jugador *jug , int c_jugo, struct
mesa_compartida *mem_comp )
```

La función levantar_cartas, devuelve un booleano (**true** o **false**), de acuerdo a si las cartas suman o no 15, y cada hijo determina el flujo de acuerdo a este resultado, de forma que el jugador debe reingresar la jugada en caso de que no pueda levantar. En caso de que pueda, se cargarán las cartas en el struct al Jugador correspondiente, en el array de cartas recolectadas.

La función descartar_carta por otro lado, solamente mueve la carta de la mano a la mesa, ya que no hay impedimento en principio a qué carta se puede descartar. El manejo de errores en este caso lo realiza el hijo, no la función (Si por ejemplo el jugador quiere descartar la carta de índice 17, que no existe en su mano.)

- **Cargar mano , mesa y escobas:**

Estas funciones son las encargadas de cargar COMO TEXTO, en el buffer utilizado para enviar mensajes, distintas cosas: La mano del jugador, las cartas en mesa, las cartas que han sido levantadas, y las escobas realizadas. De esta forma la estructura del código se simplifica bastante, se llama a la función e inmediatamente después se envía el mensaje por el socket

```
void cargar_mano(struct Jugador *jug, char *buf_tx)
void cargar_mesa(struct mesa_compartida *mem_comp,char *buf_tx)
void cargar_c_levantadas(struct Jugador *jug , char *buf_tx)
void cargar_escobas(struct Jugador *jug , char *buf_tx)
```

Estas funciones son las encargadas de cargar COMO TEXTO, en el buffer utilizado para enviar mensajes, distintas cosas: La mano del jugador, las cartas en mesa, las cartas que han sido levantadas, y las escobas realizadas. De esta forma la estructura del código se simplifica bastante, se llama a la función e inmediatamente después se envía el mensaje por el socket

Detalles de funcionamiento:

- No entraremos en detalle respecto a las primeras líneas del main, ya que son principalmente declaración de variables y utilización de funciones para inicializar, ya sea el socket, la memoria compartida, la cola de mensajes,etc. y fueron utilizadas exhaustivamente en TP's anteriores.
- Primera cosa a notar y enfatizar, es el uso de la variable "nro_jugador" y "nro_jug_c". nro_jug_c es una variable contadora que incrementa cada vez que se conecta un cliente, y como es una variable que posee el proceso padre, al crearse los hijos, éstos, que son una copia exacta (en cuanto a recursos) del proceso padre, también poseerán esta variable. De hecho, en los procesos hijos, representará el orden de conexión de los mismos:

  Se conecta jugador 1 -> nro_jug_c=1 -> Copia exacta de esta variable en proceso hijo

  Se conecta jugador 2 -> nro_jug_c=2 -> Copia exacta de esta variable en proceso hijo

  Y así sucesivamente. Por esta razón dentro de cada hijo se renombra como"nro_jugador", y es utilizada a lo largo del código.

- Haciendo uso de esta idea, se plantearon los mensajes de sincronización por cola de mensajes:

  Cuando un proceso hijo quiera acceder por ejemplo a memoria compartida, se antepone un msgrcv, cuyo id será "nro_jugador". Entonces, el padre, para que el orden de juego sea el mismo que el orden de conexión, sólo deberá enviar secuencialmente mensajes de id = 1 ,2 , 3 etc cuando corresponda.

- Al comienzo de la conexión se distingue al primer jugador (nro_jug_c = 1), avisándole que es el primer jugador y que debe ingresar la cantidad de jugadores que conformarán la partida.
- Dentro del hijo se distingue al resto de los jugadores(nro_jug_c!=1), enviándole un mensaje indicando el orden en que se conectó.
- Finalmente dentro de cada hijo se le indicará al jugador que ingrese su nombre, éste será almacenado en memoria compartida posteriormente para que el proceso padre pueda acceder a él.
- Una vez ingresado el nombre, se envía un mensaje indicando que el juego iniciará cuando se terminen de conectar todos los jugadores. Esto

lo coordina el proceso padre, que no enviará las manos hasta que todos estén conectados, y los hijos quedarán bloqueados esperando un mensaje por cola de mensajes.

- Por la naturaleza de la escoba(40 cartas, repartiendo manos de 3, y con 4 cartas iniciales en mesa), siempre se juega un número entero de rondas, por ello se loopean siempre 3 jugadas por ronda: 40-4 = 36. Y 36 es divisible por 2, 3 y 4 , que es la cantidad de jugadores permitida
- Para que cada jugador reciba las jugadas del resto, se implementaron loops con variables genéricas(dependientes del número total de jugadores y el número de jugador correspondiente a ese hijo particular), que se ejecutan ANTES, y DESPUÉS de cada jugada. De esta forma, por ejemplo, para una partida de 4 jugadores:
    - El 1er jugador no esperará ninguna jugada antes de poder realizar la suya
    - El 2do jugador esperará una sóla jugada (jugador 1), antes de realizar la suya. Y así sucesivamente.

  Y para las jugadas posteriores:

    - El 1er jugador esperará 3 jugadas luego de realizar la suya (Jugadores 2, 3 y 4)
    - El 2do jugador esperará 2 jugadas luego de realizar la suya (Jugadores 3 y 4). Y así sucesivamente.
- El handling de errores de las jugadas se implementó con booleanas que se van modificando según la respuesta que se obtenga del cliente.
- Terminado el flujo de la jugada, se recibe un mensaje del proceso padre, con un flag, si dicho flag es verdadero indica que no hay más rondas. Se evalúa quién fue el último jugador en LEVANTAR cartas de la mesa, y dicho jugador se llevará las cartas restantes de la mesa.
- La estructura del proceso padre es más sencilla, ya que consiste de loops finitos secuenciados uno atrás de otro, habilitando acceso a memoria, reenviando jugadas, indicando quién juega, y evaluando una vez terminado el flujo si quedan o no cartas por repartir.
- Por último, cabe mencionar que utilizamos un script para correr rápidamente el server y debuggearlo correctamente. El problema que encontramos al comienzo es que, como se utiliza tanto memoria compartida como cola de mensajes, si se tenía que interrumpir la ejecución por alguna razón inesperada, estas quedaban inicializadas y cargadas con datos espurios, el script que utilizamos es el siguiente:

```
#!/bin/bash

        ipcrm -M 0x7b150301

        ipcrm -Q 0x211502a8

        cd ~/Desktop/Trabajo_integrador && gcc -o p1
server_nc.c && ./p1
```

Este script elimina tanto memoria compartida como cola de mensajes, sabiendo que la key de las mismas es siempre la misma ya que la ubicación y número de referencia también lo son. Y luego se ejecuta condicionalmente un cambio de directorio -> compilación -> ejecución.

Conclusión:

En una primera instancia se diseñó un algoritmo con el juego en un solo proceso, con la finalidad de estudiar y comprender el funcionamiento del mismo y las diferentes opciones existentes para su implementación. Luego se migró este algoritmo a la implementación final con el servidor concurrente y los clientes.

Se buscó que la implementación de las cartas, las indicaciones y mensajes a los jugador sean comprensibles y claros para una mejor interpretación del juego por parte de los jugadores, ya que no se utilizó ningún tipo de interfaz gráfica o GUI. Para tal fin se utilizaron guiones, paréntesis, mayúsculas y minúsculas, saltos de línea, etc.