Grupo Clientes

Documentación

Integrantes del grupo: Andrés Ariel, Crespo Lautaro, Mariezcurrena Ivo, Mesa Joaquin, Rivero Lucia, Romero Nicolás.

Asignatura: Sistemas operativos

Año 2025

Estructura Modular del Cliente

El sistema cliente ha sido desarrollado de forma modular para separar responsabilidades y facilitar el mantenimiento del código. Cada módulo tiene una función específica en la experiencia del usuario.

Módulo	Archivo	Responsabilidad
Menú Principal	main.c	Sistema de navegación, conexión al directorio, orquestación
Selección Rol	seleccion-rol.c	Interfaz para elegir entre Explorador y Guardián
Selección Mapa	seleccion-mapa.c	Lista y selección de catacumbas disponibles
Motor de Juego	jugador.c	Lógica de juego, hilos, comunicación con servidor
Demo	base.c	Modo demostración con mapa fijo
Fin de Juego	pantalla-final.c	Hilos y pantallas de victoria, derrota y estado de directorio
Sistema Visual	colores.c/h	Gestión centralizada de colores y temas visuales

Diagrama de flujo de juego general

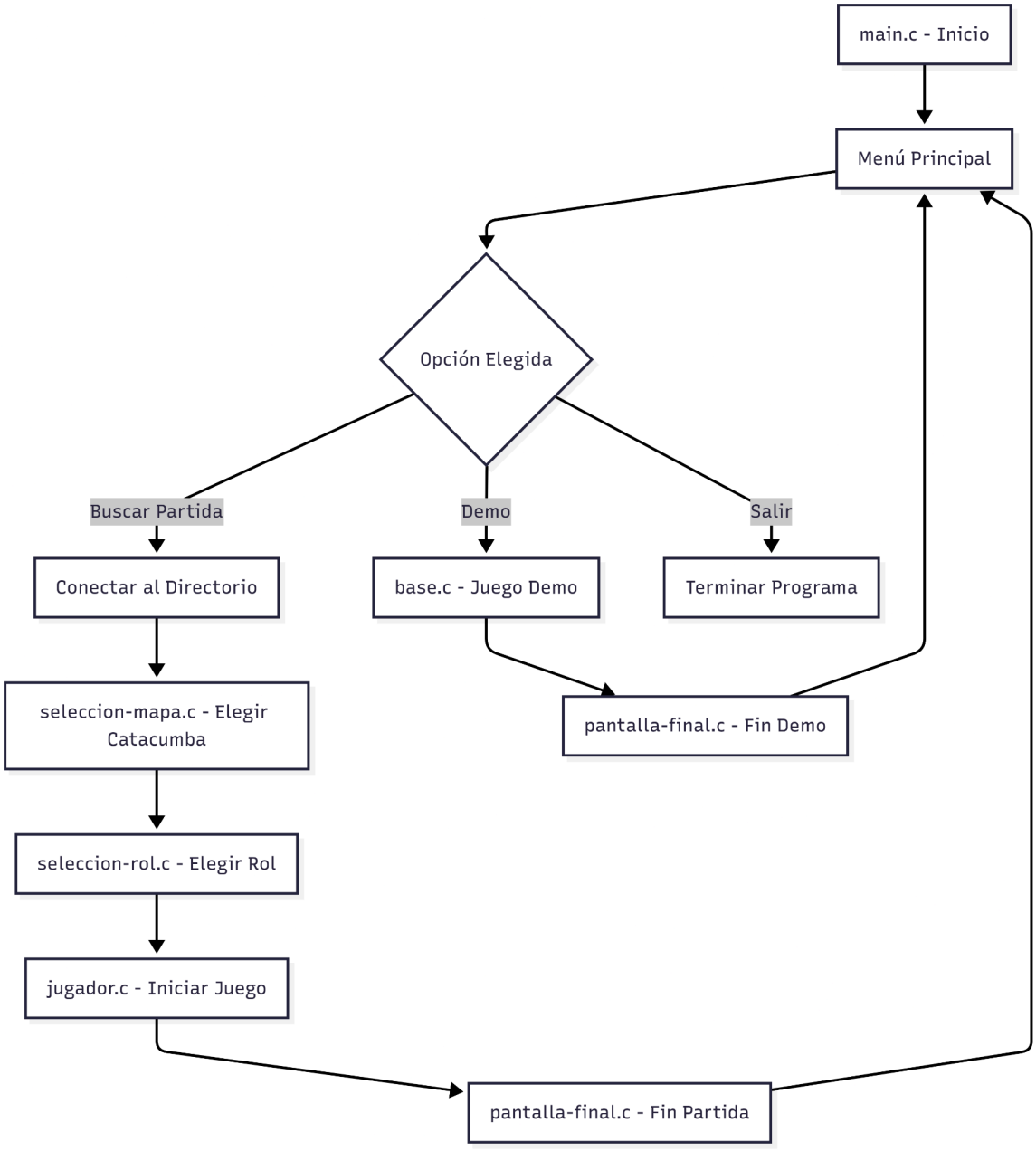

Diagrama de secuencia sobre la comunicación con otros módulos

Se adjunta a la entrega para mejorar su visualización

Comunicación con otros módulos

1. Comunicación CLIENTE ↔ DIRECTORIO

Mailboxes Utilizados

```
// Definiciones globales en directorio.h
#define MAILBOX_KEY 12345          // Solicitudes al directorio
#define MAILBOX_RESPUESTA_KEY 12346 // Respuestas del directorio

// Conexión en main.c - buscar_catacumbas_disponibles()
int mailbox_solicitudes = msgget(MAILBOX_KEY, 0666);
int mailbox_respuestas = msgget(MAILBOX_RESPUESTA_KEY, 0666);
```

Mailbox	Clave	Propósito	Dirección
Solicitudes	12345	Enviar peticiones al directorio	Cliente → Directorio
Respuestas	12346	Recibir información de catacumbas	Directorio → Cliente

Estructura de Solicitud al Directorio

```
struct solicitud {
    long mtype;          // PID del cliente (identificador único)
    int tipo;            // Tipo de operación
    char texto[MAX_TEXT]; // Datos adicionales (si son necesarios)
};

// Operaciones disponibles:
```

```
#define OP_LISTAR 1       // Solicitar lista de catacumbas
#define OP_AGREGAR 2      // Registrar nueva catacumba (usado por servidores)
#define OP_ELIMINAR 3     // Eliminar catacumba (usado por servidores)
```

Estructura de Respuesta del Directorio

```
struct respuesta {
   long mtype;          // PID del cliente destinatario
   int codigo;          // Código de resultado
   char datos[MAX_DATA]; // Información de catacumbas
};

// Códigos de respuesta:
#define RESP_OK 0         // Operación exitosa
#define RESP_ERROR 1      // Error en la operación
#define RESP_NO_HAY 2     // No hay catacumbas disponibles
```

Flujo de Comunicación con Directorio

```
int buscar_catacumbas_disponibles() {
   // 1. CONEXIÓN
   int mailbox_solicitudes = msgget(MAILBOX_KEY, 0666);
   int mailbox_respuestas = msgget(MAILBOX_RESPUESTA_KEY, 0666);

   if (mailbox_solicitudes == -1 || mailbox_respuestas == -1) {
      mostrar_pantalla_directorio_no_disponible();
      return 0;
   }

   // 2. SOLICITUD
   struct solicitud msg;
   msg.mtype = getpid();     // PID como identificador único
   msg.tipo = OP_LISTAR;     // Solicitar lista de catacumbas
   msg.texto[0] = '\0';      // Sin datos adicionales

   if (msgsnd(mailbox_solicitudes, &msg, sizeof(msg) - sizeof(long), 0) == -1) {
      perror("Error enviando solicitud");
      return 0;
   }

   // 3. RESPUESTA
   struct respuesta resp;
   if (msgrcv(mailbox_respuestas, &resp, sizeof(resp) - sizeof(long),
```

```
        getpid(), 0) == -1) {
    perror("Error recibiendo respuesta");
    return 0;
}

// 4. PROCESAMIENTO
if (resp.codigo == RESP_OK) {
    // Parsear datos en formato: "nombre|direccion|mailbox|jugadores|max;..."
    char *saveptr1, *saveptr2;
    char *catacumba = strtok_r(resp.datos, ";", &saveptr1);

    while (catacumba && num_maps < MAX_MAPS) {
        char *nombre = strtok_r(catacumba, "|", &saveptr2);
        char *direccion = strtok_r(NULL, "|", &saveptr2);
        char *mailbox_str = strtok_r(NULL, "|", &saveptr2);
        char *cantJug = strtok_r(NULL, "|", &saveptr2);
        char *maxJug = strtok_r(NULL, "|", &saveptr2);

        // Almacenar en estructura Map
        strcpy(maps[num_maps].name, nombre);
        strcpy(maps[num_maps].shm_path, direccion);
        maps[num_maps].mailbox = atoi(mailbox_str);
        maps[num_maps].players_connected = atoi(cantJug);
        maps[num_maps].max_players = atoi(maxJug);
        num_maps++;

        catacumba = strtok_r(NULL, ";", &saveptr1);
    }

    // 5. SELECCIÓN DE MAPA
    int seleccionado = mostrar_seleccion_mapa(maps, num_maps);
    if (seleccionado >= 0) {
        // Almacenar configuración seleccionada
        strcpy(selected_shm_path, maps[seleccionado].shm_path);
        selected_mailbox = maps[seleccionado].mailbox;

        // Continuar con selección de rol
        int rol_seleccionado = mostrar_menu_rol();
        if (rol_seleccionado >= 0) {
            set_game_role(“rol_selecionado”);
            setPlayChar(rol_selecionado);
        }
        jugar();  // Iniciar el juego
    }
```

```
    }

    return 0;
}
```

2. Comunicación CLIENTE ↔ SERVIDOR DE CATACUMBAS

Mailboxes Dinámicos

```
// Variables globales para comunicación con servidor
int selected_mailbox;                // ID del mailbox del servidor (del directorio)
key_t key_respuestas_global;         // Clave única del cliente
int mailbox_respuestas_global;       // Mailbox personal del cliente
int mailbox_solicitudes_id_global;   // Mailbox del servidor
```

Configuración de Mailboxes

```
int conectar_al_servidor() {
    // 1. Crear mailbox único para respuestas del cliente
    key_t key_respuestas = ftok("/tmp", getpid());  // Clave basada en PID
    key_respuestas_global = key_respuestas;

    mailbox_respuestas_global = msgget(key_respuestas, IPC_CREAT | 0666);
    if (mailbox_respuestas_global == -1) {
        perror("Error creando mailbox respuestas");
        return -1;
    }

    // 2. Conectar al mailbox del servidor (obtenido del directorio)
    mailbox_solicitudes_id_global = msgget(selected_mailbox, 0666);
    if (mailbox_solicitudes_id_global == -1) {
        perror("Error conectando al servidor");
        return -1;
    }

    return 0;
}
```

Estructura de Solicitud al Servidor

```
struct SolicitudServidor {
    long mtype;                  // PID del cliente
```

```
    int codigo;                 // Tipo de solicitud
    key_t clave_mailbox_respuestas; // Para que el servidor responda
    int fila, columna;          // Posición del jugador
    char tipo;                  // RAIDER o GUARDIAN
};
```

Códigos de Solicitud

Código	Valor	Descripción	Cuándo se Envía
CONEXION	1	Conectar jugador al servidor	Al iniciar partida
MOVIMIENTO	2	Mover jugador en el mapa	Por cada movimiento válido
TESORO_CAPTURADO	3	Notificar captura de tesoro	Al recoger tesoro
RAIDER_CAPTURADO	4	Notificar captura de raider	Cuando guardián atrapa raider
DESCONEXION	5	Desconectar del servidor	Al salir del juego

Estructura de Respuesta del Servidor

```
struct RespuestaServidor {
    long mtype;                 // PID del cliente destinatario
    int codigo;                 // Código de respuesta
    char mensaje[MAX_LONGITUD_MENSAJES]; // Mensaje descriptivo
};
```

Códigos de Respuesta

Código	Valor	Significado	Acción del Cliente
S_OK	0	Operación exitosa	Continuar juego normalmente
MUERTO	1	Jugador eliminado	Mostrar pantalla game over

SIN_RAIDERS	2	Victoria guardianes	Mostrar pantalla victoria guardian
SIN_TESOROS	3	Victoria raiders	Mostrar pantalla victoria tesoros
ERROR	-1	Error en operación	Mostrar mensaje error

Flujo de Comunicación con Servidor

1. Conexión Inicial

```
int enviar_solicitud_conexion() {
    struct SolicitudServidor solicitud;
    solicitud.mtype = getpid();
    solicitud.codigo = CONEXION;
    solicitud.clave_mailbox_respuestas = key_respuestas_global;
    solicitud.fila = jugador_y_global;
    solicitud.columna = jugador_x_global;
    solicitud.tipo = player_character;  // RAIDER o GUARDIAN

    // Enviar solicitud (bloqueante)
    if (msgsnd(mailbox_solicitudes_id_global, &solicitud,
            sizeof(solicitud) - sizeof(long), 0) == -1) {
        perror("Error enviando solicitud de conexión");
        return -1;
    }

    // Esperar confirmación
    struct RespuestaServidor respuesta;
    if (msgrcv(mailbox_respuestas_global, &respuesta,
            sizeof(respuesta) - sizeof(long), getpid(), 0) == -1) {
        perror("Error recibiendo confirmación");
        return -1;
    }

    return (respuesta.codigo == S_OK) ? 0 : -1;
}
```

2. Envío de Movimientos

```
void enviar_movimiento_al_servidor(int jugador_x, int jugador_y,
```

```
                        key_t clave_mailbox_respuestas,
                        int mailbox_solicitudes_id) {
    struct SolicitudServidor solicitud;
    solicitud.mtype = getpid();
    solicitud.codigo = MOVIMIENTO;
    solicitud.clave_mailbox_respuestas = clave_mailbox_respuestas;
    solicitud.fila = jugador_y;
    solicitud.columna = jugador_x;
    solicitud.tipo = player_character;

    // Envío no bloqueante para evitar lag
    if (msgsnd(mailbox_solicitudes_id, &solicitud,
            sizeof(solicitud) - sizeof(long), IPC_NOWAIT) == -1) {
        perror("Error enviando movimiento");
    }
}
```

3. Captura de Tesoros

```
void enviar_captura_tesoro_al_servidor(int jugador_x, int jugador_y,
                        key_t clave_mailbox_respuestas,
                        int mailbox_solicitudes_id) {
    struct SolicitudServidor solicitud;
    solicitud.mtype = getpid();
    solicitud.codigo = TESORO_CAPTURADO;
    solicitud.clave_mailbox_respuestas = clave_mailbox_respuestas;
    solicitud.fila = jugador_y;
    solicitud.columna = jugador_x;
    solicitud.tipo = player_character;

    if (msgsnd(mailbox_solicitudes_id, &solicitud,
            sizeof(solicitud) - sizeof(long), IPC_NOWAIT) == -1) {
        perror("Error enviando captura tesoro");
    }
}
```

4. Recepción de Eventos Asíncronos

```
// En hilo_entrada() - Verificación no bloqueante de eventos
void *hilo_entrada(void *arg) {
    while (running) {
        // ... procesamiento de entrada del usuario ...

        // Verificar eventos del servidor (no bloqueante)
```

```
        struct RespuestaServidor evento;
        if (msgrcv(mailbox_respuestas_global, &evento,
                sizeof(evento) - sizeof(long), getpid(), IPC_NOWAIT) != -1) {

            switch (evento.codigo) {
                case MUERTO:
                    running = 0;
                    mostrar_game_over();
                    break;

                case SIN_RAIDERS:
                    running = 0;
                    mostrar_pantalla_victoria_guardian();
                    break;

                case SIN_TESOROS:
                    running = 0;
                    mostrar_pantalla_victoria_tesoros();
                    break;

                case S_OK:
                    // Movimiento confirmado, continuar
                    break;

                default:
                    fprintf(stderr, "Evento desconocido: %d\n", evento.codigo);
            }
        }
    }
    return NULL;
}
```

5. MEMORIA COMPARTIDA

Configuración de Memoria Compartida

```
// Variables globales para memoria compartida
char *mapa = NULL;              // Puntero al mapa compartido
int fd = -1;                    // Descriptor de archivo
char selected_shm_path[128];    // Ruta proporcionada por el servidor
```

Inicialización

```
int inicializar_memoria_mapa() {
```

```
    // 1. Abrir memoria compartida creada por el servidor
    fd = shm_open(selected_shm_path, O_RDWR, 0666);
    if (fd == -1) {
        perror("Error abriendo memoria compartida");
        return -1;
    }

    // 2. Mapear memoria en el espacio de direcciones del proceso
    mapa = mmap(NULL, FILAS * COLUMNAS, PROT_READ | PROT_WRITE,
                MAP_SHARED, fd, 0);
    if (mapa == MAP_FAILED) {
        perror("Error mapeando memoria");
        close(fd);
        return -1;
    }

    printf("Memoria compartida mapeada exitosamente en: %p\n", mapa);
    return 0;
}
```

Uso de la Memoria Compartida

Lectura del Mapa (Hilo de Renderizado)

```
void dibujar_mapa_coloreado() {
    // Acceso directo a la memoria compartida
    for (int y = 0; y < FILAS; y++) {
        for (int x = 0; x < COLUMNAS; x++) {
            char c = mapa[y * COLUMNAS + x];  // Lectura directa

            // Destacar jugador local
            if (y == jugador_y_global && x == jugador_x_global) {
                wattron(ventana_mapa, COLOR_PAIR(COLOR_JUGADOR_LOCAL) | A_BOLD);
                mvwaddch(ventana_mapa, y + 4, x + 2, 'J');
                wattroff(ventana_mapa, COLOR_PAIR(COLOR_JUGADOR_LOCAL) | A_BOLD);
            }
            // Renderizar elementos del mapa
            else {
                switch (c) {
                    case PARED:
                        wattron(ventana_mapa, COLOR_PAIR(COLOR_PARED));
                        mvwaddch(ventana_mapa, y + 4, x + 2, c);
                        wattroff(ventana_mapa, COLOR_PAIR(COLOR_PARED));
                        break;
```

```
                case TESORO:
                    wattron(ventana_mapa, COLOR_PAIR(COLOR_TESORO));
                    mvwaddch(ventana_mapa, y + 4, x + 2, c);
                    wattroff(ventana_mapa, COLOR_PAIR(COLOR_TESORO));
                    break;

                case RAIDER:
                    wattron(ventana_mapa, COLOR_PAIR(COLOR_RAIDER) | A_BOLD);
                    mvwaddch(ventana_mapa, y + 4, x + 2, c);
                    wattroff(ventana_mapa, COLOR_PAIR(COLOR_RAIDER) | A_BOLD);
                    break;

                case GUARDIAN:
                    wattron(ventana_mapa, COLOR_PAIR(COLOR_GUARDIAN) | A_BOLD);
                    mvwaddch(ventana_mapa, y + 4, x + 2, c);
                    wattroff(ventana_mapa, COLOR_PAIR(COLOR_GUARDIAN) | A_BOLD);
                    break;

                case VACIO:
                default:
                    wattron(ventana_mapa, COLOR_PAIR(COLOR_PISO));
                    mvwaddch(ventana_mapa, y + 4, x + 2, c);
                    wattroff(ventana_mapa, COLOR_PAIR(COLOR_PISO));
                    break;
            }
        }
    }
  }

  wnoutrefresh(ventana_mapa);
  doupdate( );
}
```

Validación Local de Movimientos

```
int procesar_movimiento(char destino, int *jugador_x, int *jugador_y,
                        int new_x, int new_y) {
  // Verificar límites del mapa
  if (new_x < 0 || new_x >= COLUMNAS || new_y < 0 || new_y >= FILAS) {
      return 0;  // Movimiento inválido
  }

  // Leer directamente de memoria compartida
  char destino_char = mapa[new_y * COLUMNAS + new_x];
```

```
    // Verificar colisión con pared
    if (destino_char == PARED) {
        return 0;  // No se puede mover
    }

    // Detectar eventos especiales
    if (destino_char == TESORO && player_character == RAIDER) {
        // Notificar al servidor sobre captura de tesoro
        enviar_captura_tesoro_al_servidor(new_x, new_y, key_respuestas_global,
                                          mailbox_solicitudes_id_global);
    }

    // Actualizar posición local
    *jugador_x = new_x;
    *jugador_y = new_y;

    return 1;  // Movimiento válido
}
```

Multihilos y Sincronización

Hilos Principales del Sistema

```
// Variables globales para hilos
pthread_t thread_refresco;          // Hilo de renderizado
pthread_t thread_entrada;           // Hilo de entrada del usuario
pthread_mutex_t pos_mutex;          // Mutex para proteger posiciones
volatile int running = 1;           // Control global de hilos
```

Hilo de Renderizado

```
void *hilo_refresco(void *arg) {
    while (running) {
        // Renderizado continuo del mapa
        pthread_mutex_lock(&pos_mutex);
        dibujar_mapa_coloreado();
        pthread_mutex_unlock(&pos_mutex);

        // Control de frecuencia (20 FPS)
        usleep(50000);  // 50ms
    }
    return NULL;
}
```

Hilo de Entrada

```
void *hilo_entrada(void *arg) {
    int ch;
    while (running && (ch = getch()) != 'q') {
        int new_x = jugador_x_global;
        int new_y = jugador_y_global;

        // Procesar teclas de dirección
        switch (ch) {
            case KEY_UP:    new_y--; break;
            case KEY_DOWN:  new_y++; break;
            case KEY_LEFT:  new_x--; break;
            case KEY_RIGHT: new_x++; break;
            default: continue;
        }

        // Validar y procesar movimiento
```

```
        pthread_mutex_lock(&pos_mutex);
        if (procesar_movimiento(mapa[new_y * COLUMNAS + new_x],
                                &jugador_x_global, &jugador_y_global,
                                new_x, new_y)) {
            // Enviar movimiento válido al servidor
            enviar_movimiento_al_servidor(jugador_x_global, jugador_y_global,
                                          key_respuestas_global,
                                          mailbox_solicitudes_id_global);
        }
        pthread_mutex_unlock(&pos_mutex);

        // Verificar eventos del servidor (no bloqueante)
        verificar_eventos_servidor();
    }

    running = 0;  // Señalar terminación
    return NULL;
}
```

Sincronización de Hilos

```
// Función principal del juego
void jugar() {
    // Inicializar mutex
    pthread_mutex_init(&pos_mutex, NULL);

    // Configurar ncurses para uso concurrente
    initscr();
    inicializar_colores();
    noecho();
    cbreak();
    keypad(stdscr, TRUE);
    curs_set(0);

    // Conectar al servidor y inicializar memoria
    if (conectar_al_servidor() != 0 || inicializar_memoria_mapa() != 0) {
        terminarPartida();
        return;
    }

    // Crear ventana para el mapa
    if (ventana_mapa == NULL) {
        ventana_mapa = newwin(FILAS + 10, COLUMNAS + 10, 0, 0);
    }
```

```
    // Crear hilos especializados
    pthread_create(&thread_refresco, NULL, hilo_refresco, NULL);
    pthread_create(&thread_entrada, NULL, hilo_entrada, NULL);

    // Esperar terminación del hilo de entrada (usuario presiona 'q')
    pthread_join(thread_entrada, NULL);
    running = 0;

    // Esperar terminación del hilo de renderizado
    pthread_join(thread_refresco, NULL);

    // Limpieza de recursos
    pthread_mutex_destroy(&pos_mutex);
    terminarPartida();
}
```

Gestión de Recursos

Limpieza Automática

```
int recursos_limpiados = 0;  // Flag para evitar doble limpieza

void terminarPartida() {
    if (recursos_limpiados) return;

    // 1. Notificar desconexión al servidor
    if (mailbox_solicitudes_id_global != -1) {
        struct SolicitudServidor solicitud;
        solicitud.mtype = getpid();
        solicitud.codigo = DESCONEXION;
        solicitud.clave_mailbox_respuestas = key_respuestas_global;
        solicitud.fila = jugador_y_global;
        solicitud.columna = jugador_x_global;
        solicitud.tipo = player_character;

        // Envío no bloqueante (el servidor puede estar cerrado)
        msgsnd(mailbox_solicitudes_id_global, &solicitud,
               sizeof(solicitud) - sizeof(long), IPC_NOWAIT);
    }

    // 2. Limpiar memoria compartida
    if (mapa != NULL && mapa != MAP_FAILED) {
        munmap(mapa, FILAS * COLUMNAS);
        mapa = NULL;
    }

    if (fd != -1) {
        close(fd);
        fd = -1;
    }

    // 3. Limpiar mailboxes
    if (mailbox_respuestas_global != -1) {
        msgctl(mailbox_respuestas_global, IPC_RMID, NULL);
        mailbox_respuestas_global = -1;
    }

    // 4. Limpiar ncurses
    if (ventana_mapa != NULL) {
```

```
        delwin(ventana_mapa);
        ventana_mapa = NULL;
    }

    endwin();

    // 5. Marcar recursos como limpiados
    recursos_limpiados = 1;

    printf("Recursos limpiados exitosamente\n");
}

// Función de desconexión explícita
void desconectar_del_servidor() {
    terminarPartida();
}
```

Manejo de Señales

```
void configurar_senales() {
    // Manejar SIGINT (Ctrl+C) para limpieza correcta
    signal(SIGINT, signal_handler);
    signal(SIGTERM, signal_handler);
}

void signal_handler(int sig) {
    printf("\nSeñal %d recibida, limpiando recursos...\n", sig);
    running = 0;
    terminarPartida();
    exit(0);
}
```

Eventos Especiales

Detección de Eventos de Fin de Juego

```
void verificar_eventos_servidor() {
    struct RespuestaServidor evento;

    // Verificación no bloqueante
    if (msgrcv(mailbox_respuestas_global, &evento,
               sizeof(evento) - sizeof(long), getpid(), IPC_NOWAIT) != -1) {

        switch (evento.codigo) {
            case MUERTO:
                running = 0;
                // Mostrar pantalla de game over con mensaje personalizado
                mostrar_game_over_con_mensaje(evento.mensaje);
                break;

            case SIN_RAIDERS:
                running = 0;
                // Victoria de los guardianes
                mostrar_pantalla_victoria_guardian_con_mensaje(evento.mensaje);
                break;

            case SIN_TESOROS:
                running = 0;
                // Victoria de los raiders
                mostrar_pantalla_victoria_tesoros_con_mensaje(evento.mensaje);
                break;

            case S_OK:
                // Operación exitosa, continuar normalmente
                break;

            case ERROR:
                // Error del servidor, mostrar mensaje pero continuar
                mvprintw(FILAS + 8, 0, "Error del servidor: %s", evento.mensaje);
                refresh();
                break;

            default:
                // Evento desconocido
                fprintf(stderr, "Evento desconocido del servidor: %d\n", evento.codigo);
```

```
    }
  }
}
```

Códigos de mensajes

Evento	Código	Origen	Resultado	Pantalla Mostrada
Tesoro Capturado	TESORO_CAPTURADO	Cliente → Servidor	Actualización del mapa	Continúa juego
Raider Capturado	RAIDER_CAPTURADO	Servidor → Cliente	Eliminación del raider	mostrar_game_over()
Sin Raiders	SIN_RAIDERS	Servidor → Todos	Fin del juego	mostrar_pantalla_victoria_guardian()
Sin Tesoros	SIN_TESOROS	Servidor → Todos	Fin del juego	mostrar_pantalla_victoria_tesoros()
Error Fatal	ERROR	Servidor → Cliente	Mensaje de error	Mensaje en pantalla