Relazione Esame Reti

Luca Maiuri

December 2024

Sommario

1 Prefazione

In questa breve prefazione si vuole dare un idea dell'approccio che si è seguito nella stesura di questo progetto, partendo come base la traccia data:

"La navicella (client) entra nel settore spaziale dei meteoriti (server) connettendosi in UDP. La tempesta di meteoriti genera in modo casuale n pacchetti UDP ogni 2 secondi che rappresentano i detriti in una griglia MxM. La navicella riceve un alert nel caso in cui il detrito spaziale si trovi sulla sua stessa posizione e può spostarsi prima dell'impatto"

Nel mio progetto quindi avevo i seguenti requisiti:

- Pacchetti UDP: connessioni socket UDP.
- N pacchetti ogni 2 secondi: si opta per 1 pacchetto ogni 2 secondi di grandezza prefissata.
- Griglia di gioco MxM: si è scelta una griglia di 11x20, 1 riga di controllo per messaggi.
- Messaggio di alert: il messaggio compare/scompare nell'undicesima riga che è usata anche per altri messaggi. Il messaggio comparirà se un meteorite è ad un distanza verticale di 4 cella dalla nave.

Questi sono i requisiti imposti dalla traccia, vi sono però delle necessità implementative che si sono presentate man mano che si delineava il progetto, i requisiti aggiuntivi sono i seguenti:

- Il server per quanto sia connesso in UDP, deve avere una lista di client a cui inviare i datagrammi che genera. Questa lista va aggiornata di volta in volta quando i dati non raggiungono più il client o ne arriva uno nuovo. Avremo quindi un massimo numero di client servibili.
- Il client deve in qualche modo capire se il server a cui si rivolge è ancora attivo, questo richiede di catturare l'errore asincrono ICMP relativo all'invio dei dati.
- Il gioco in se stesso richiede un sotto-processo a parte generato dal client e con esso si scambia dati sull'andamento del gioco e sulla sua fine.

Infine si vuole aggiungere che durante la stesura del progetto si è fatto ampiamente uso delle attuali algoritmi generativi forniti da servizi di GitHub-Copilot, in particolare sono stati usati i modelli LLM di : OpenAi 4o e Claude sonnet.

Ci si tiene a sottolineare che il progetto **non è stato costruito** da questi strumenti, ma sono stati utilizzati per vari scopi per coadiuvare lo sviluppo ed accelerarne la stesura, esempi di utilizzo:

- Chiedere informazioni su librerie, funzioni, macro ed errori laddove non si conoscevano i dettagli.
- Chiedere un parare implementativo: "Come realizzare uno scambio di dati IPC?" oppure "Come aprire un terminale attraverso il linguaggio C?" e simili.
- Auto completamento delle parti ripetitive, commenti, cicli, sintassi di base.
- Suggerimenti per compilazioni di funzioni, blocchi di codice logici "if" , condizioni di errori, debugging attraverso la nuova funzione di debugging integrato fra Copilot e VSCODE.

Ogni richiesta è stata sempre analizzata e rivista, si è sempre poi cercato la motivazione delle risposte ricevute e si è consultata la documentazione relativa (Ad esempio: man7.org) ciò ha portato spesso a scartare i suggerimenti dati essendo palesemente errati, ci tengo infatti a mostrare qui il seguente errore su un prompt a GitHub Copilot sulla possibilità di monitorare con Epoll i file descriptor delle code messaggi POSIX, e con questo concludo la prefazione.

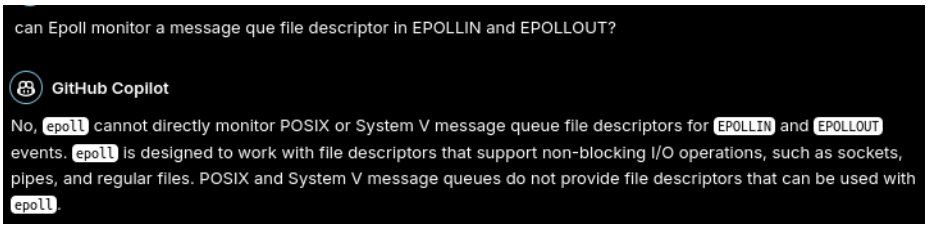

Figure 1: Prompt a copilot chat.

Linux implementation of message queue descriptors
On Linux, a message queue descriptor is actually a file descriptor. (POSIX does not require such an implementation.) This means that a message queue descriptor can be monitored using select(2), poll(2), or epoll(7). This is not portable.

The close-on-exec flag (see open(2)) is automatically set on the file descriptor returned by mq_open(2).

Figure 2: Manuale man7 delle code POSIX

2 Descrizione e schema dell'architettura

Il progetto nella mia idea si compone di tre parti che rappresentano tre processi distinti che operano in tre schermate di terminali diversi, rispettivamente : client, server e motore di gioco.

Per realizzare il progetto si è optato per il linguaggio C, in ambiente Unix sotto una distribuzione di linux (ZorinOS) con terminale gnome terminal. L'ambiente di sviluppo è VSCODE text editor con estensioni di LLM GitHub copilot e CMAKE.

Sia server che client si basano su socket UDP, i socket sono monitorati da un monitor di file descriptor: Epoll, essa è un API che deriva da Poll.h, similmente alla select ma con molti meno passaggi implementativi, limiti sui file descriptor e prestazioni migliori rispetto alla semplice select. Il client e server usano la Epoll per monitorare i socket, nel client in aggiunta vengono monitorate le code di messaggi per lo scambio di dati con il motore di gioco.

Il motore di gioco è un intero processo che vive nel proprio terminale che nasce qualora il client riceva con successo dati dal server, il gioco si basa su aggiornamenti in tempo reale dal client monitorando la coda messaggi e il file descriptor che gestisce l'input da tastiera nel terminale, con le dovute modifiche di cui si mostrerà più avanti nel dettaglio.

Di seguito uno schema a blocchi del progetto e di come i tre elementi comunicano fra di loro, si imita lo stile degli schemi delle slide del corso.

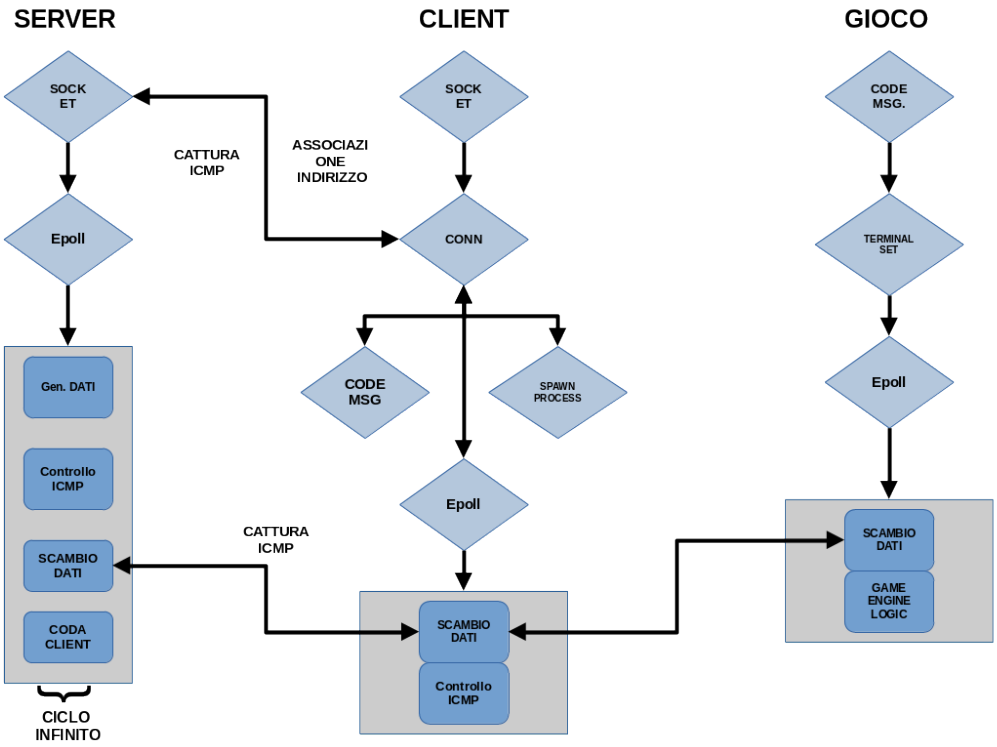

Figure 3: Schema generale

2.1 Server: implementazione

Il server segue la seguente logica implementativa:

- Check degli argomenti usati per lanciare il programma.
- Creazione del socket e impostazioni delle opzioni e flag, uguali per client e server:
  - SO_REUSEADDR: riutilizza l'indirizzo locale.
  - SO_RCVTIMEO: timeout per la ricezione messaggi.
  - SO_IP_RECVERR: Opzione di livello SOL_IP mentre le precedenti erano proprie del livello SOL_SOCKET, permette di catturare i messaggi errore ICMP.
- Generazione di una porta casuale effimera.
- Bind dell'indirizzo e porta del server al socket creato.
- Creazione istanza Epoll, aggiunta del file descriptor del socket al pool di file descriptor monitorati, modalità EPOLLIN e OUT.
- Ciclo infinito ,in cui vengono monitorati gli eventi Epoll ed a seconda degli eventi il server reagisce ed applica determinate azioni.
- Se un client manda un messaggio specifico, il server monitorando il socket in lettura controlla il messaggio ed in caso affermativo aggiunge il client alla sua coda di invio alla prossima iterazione.
- Sleep di due secondi prima di generare un nuovo set di meteoriti che saranno inviati sul socket se pronto in scrittura.

La problematica principale del server è quella di tenere traccia di chi si connette e a chi mandare i messaggi, per quanto UDP sia un protocollo connectionless la funzione di invio dati *sendTo* necessita comunque di un indirizzo di destinatario, per cui vengono mantenuti in una array di massimo 50 client le informazioni relative ai client che sono arrivati al server per chiedere le posizioni dei meteoriti.

```
// Inizializzazione della struttura per la ricezione dell'indirizzo del client da
↪  messaggi
    struct sockaddr_in clientAddr[MAX_CLIENTS];
    char *dummyIPdecimals = "255.255.255.255";
    for (int i = 0; i < MAX_CLIENTS; i++)
    {
        // Metto tutti gli indirizzi a un valore non valido per i miei scopi
        setGenericIPV4(dummyIPdecimals, &clientAddr[i].sin_addr);
        clientAddr[i].sin_port = htons(0);
    }
```

Il codice sopra mostra come questa coda è strutturata, si crea un array da 50 client il cui tipo è la struttura dati **sockaddr_in** in cui inizialmente vengono inizializzati valori inutilizzabili che servono solamente a soddisfare una logica di controllo per gestire l'inserimento e l'uscita dei client.

```
if (fdEvents->events & EPOLLIN)
        {
            // Controllo chi è il client che ha inviato il messaggio
            struct sockaddr_in currentClient;
            customRecvFrom(serverSocket, messageBuffer, &currentClient);  //
            ↪  Ricevo il messaggio
            char *currentAddrComplete = getAddressString(&currentClient); //
            ↪  Estraggo l'indirizzo completo in formato "IP:PORT"
            printf("[INFO] Messaggio ricevuto: %s da %s\n", messageBuffer,
            ↪  currentAddrComplete);

            // Controllo che messaggio ho ricevuto, nel caso di START accetto
            ↪  il client
            if (strcmp(messageBuffer, "START") == 0)
            {
                // Vedo se lo devo aggiungere alla lista dei client
                int clientAlreadyInList = 0;
                for (int j = 0; j < MAX_CLIENTS; j++)
                {
                    // Il confronto include le porte.
                    char *arrayAddr = getAddressString(&clientAddr[j]);
                    if (strcmp(arrayAddr, currentAddrComplete) == 0)
                    {
                        clientAlreadyInList = 1;
                        printf("[INFO] Client già in lista\n");
                        free(arrayAddr);
                        break;
                    }
                    free(arrayAddr);
                }
                if (clientAlreadyInList == 0)
                {
                    printf("[INFO] Aggiungo il client alla lista\n");

                    // Trovo il primo slot libero per il client
                    for (int h = 0; h < MAX_CLIENTS; h++)
                    {

                        // Confronto con l'indirizzo presente nella struttura
                        if (clientAddr[h].sin_port == 0)
                        {
                            clientAddr[h] = currentClient;
                            printf("[INFO] Client aggiunto alla lista in
                            ↪  posizione %d\n", h);
                            break;
                        }
                    }
                }
```

```
                // Scarico il buffer del messaggio del client
                freeUDPMessage(messageBuffer);
                messageBuffer = getStdUDPMessage();
            }
            free(currentAddrComplete);
        }
```

In questo frammento si vede la logica di inserimento dei client all'interno della coda in caso di evento EPOLLIN sul file descriptor del Server, vengono memorizzate le informazioni dei Client che mandano un messaggio del tipo : "START" nel primo slot libero che è rappresentato dagli indirizzi inutilizzabili precedenti. Quando è possibile mandare messaggi ai client attraverso il socket (EPOLLOUT) la lista viene di nuovo analizzata dal server, se un client a cui tenta di inviare un messaggio non risponde perché irraggiungibile(ICMP type 3 code 3, port unreachable) allora è da considerarsi non più in ascolto per cui lo si rimuove dalla lista.

```
if (fdEvents->events & EPOLLOUT)
        {
            // Invio le posizioni, se un client in lista non è più
            // valido lo rimuovo.
            for (int j = 0; j < MAX_CLIENTS; j++)
            {
                // Estraggo l'indirizzo IP del client dall'array di strutture
                char *arrayAddr = getAddressString(&clientAddr[j]);

                // Controllo se effettivamente c'è un client in lista da poter
                ↪  inviare il messaggio
                if (strcmp(arrayAddr, "255.255.255.255:0") == 0)
                {
                    free(arrayAddr);
                    continue;
                }
                else
                {
                    printf("[INFO] Invio posizioni dei meteoriti a %s\n",
                    ↪  arrayAddr);
                    // Invio delle posizioni dei meteoriti al client
                    if (customSendTo(serverSocket, meteoritesBuffer,
                    ↪  &clientAddr[j]) == 1)
                    {
                        // Rendo il client della lista inutilizzabile per le
                        ↪  future comunicazioni se non raggiungibile
                        printf("[INFO] Il client %s non è più raggiungibile,
                        ↪  rimozione.\n", arrayAddr);
                        clientAddr[j].sin_family = AF_UNSPEC;
                        setGenericIPV4("255.255.255.255",
                        ↪  &clientAddr[j].sin_addr);
                        clientAddr[j].sin_port = htons(0);
                    }
                    else
```

```
                    {
                        printf("[INFO] Posizioni inviate con successo a:
                        ↪  %s\n", arrayAddr);
                    }
                }
                free(arrayAddr);
            }
        }
```

2.1.1 Epoll

Si apre un capitolo a parte per spiegare perché si è scelto di usare Epoll e non la select. Il vantaggio principale di utilizzare Epoll sono le prestazioni, il monitor di Epoll non risente delle limitazioni sui file descriptor di select, ed ha prestazioni migliore all'aumentare dei file descriptor monitorati.

Considerando che il server idealmente dovrebbe servire molti più di 50 client previsti dal mio progetto, l'Epoll è la scelta migliore anche sulla sua originale funzione Poll da cui deriva. Infine l'usabilità di Epoll a livello di codice scritto è molto minore rispetto alla select, i file descriptor sono passati al monitor dell'istanza dell'Epoll e non vanno monitorati e resettati manualmente con le macro ad ogni evento rispetto alla select, di questi aspetti se ne occupa l'Epoll.

Similarmente alla select L'Epoll monitora i file descriptor in base agli eventi che essi generano, essa mantiene una struttura dati **epoll_event** in cui vengono inserite due tipologie di flag: EPOLLIN e EPOLLOUT che indicano al monitor per quali tipo di evento la Epoll deve catturare il thread in cui sta monitorando il file descriptor.

```
    /*
    Funzione per l'aggiunta di un file descriptor alla lista di quelli da
    ↪  monitorare
    dall'istanza di epoll.

    @param fileDescToWatch: file descriptor da monitorare
    @param eventFlags: flag per la gestione degli eventi di I/O
*/
extern void addFileDescriptorToThePolling(int fileDescToWatch, uint32_t
↪  eventFlags)
{
    // Controllo se il file descriptor è stato inizializzato, naive ma necessario.
    if (!isEpollInitialized())
    {
        customErrorPrinting("Errore: epoll non inizializzato\n");
        exit(EXIT_FAILURE);
    }

    // Inizializzazione della struttura per la gestione degli eventi
    struct epoll_event eventType;
    eventType.data.fd = fileDescToWatch; // File descriptor da monitorare
    eventType.events = eventFlags;       // Eventi da monitorare
```

```
    // Aggiunta del file descriptor alla lista di quelli da monitorare
    if (epoll_ctl(epollFd, EPOLL_CTL_ADD, fileDescToWatch, &eventType) == -1)
    {
        customErrorPrinting("Errore: epoll_ctl aggiunta FD fallita\n");
        exit(EXIT_FAILURE);
    }
    // Incremento del contatore interno di file descriptor monitorati
    fileDescriptorCounter++;
}
```

Una volta forniti i file descriptor e gli eventi su cui si deve porre attenzione, l'epoll viene messa in "attesa" della notifica di un evento, così come venivano controllati descriptor nell'insieme FD_ISSET per la select. L'attesa è inizializzata con la funzione *epoll_wait* essa prende in considerazione una struttura vuota di tipo **epoll_events** di una certa grandezza, pari agli eventi che ci si aspetta di ricevere in prospettiva, l'Epoll riempie questa lista man mano che si palesa un evento.

```
    extern int waitForEvents(struct epoll_event *eventsArray, int MAX_EVENTS)
{
    // Controllo se Epoll è stato inizializzato
    if (!isEpollInitialized())
    {
        customErrorPrinting("Errore: epoll non inizializzato\n");
        exit(EXIT_FAILURE);
    }
    // Controllo se ci sono file descriptor da monitorare
    if (fileDescriptorCounter == 0)
    {
        customErrorPrinting("Errore: nessun file descriptor da monitorare\n");
        exit(EXIT_FAILURE);
    }

    // Attesa degli eventi di I/O
    int returnedEvents = epoll_wait(epollFd, eventsArray, MAX_EVENTS, -1);
    if (returnedEvents == -1)
    {
        customErrorPrinting("[ERROR] Errore nell'attesa degli eventi\n");
        exit(EXIT_FAILURE);
    }

    return returnedEvents;
}
```

L'ulimo argomento della Epoll è un **timeout** questo timeout a seconda di come è stato impostato rende l'Epoll *bloccatne* oppure *non-bloccante*, l'Epoll a seconda del valore del timeout blocca l'esucuzione del thread dove è stata chiamata per *timeout* secondi (approssimati dal kernel) qualora nessuno dei file descriptor sia pronto la funzione rilascia il thread chiamante,esaurito il timeout, e lascia che l'esecuzione continui.

Specificare un timeout pari a zero rende la Epoll non bloccante in quanto ritorna subito se non

ci sono file descriptor pronti, infine ponendo il timeout ad un valore negativo rendiamo la Epoll completamente *bloccante in maniera indefinita* ossia finché un file descriptor non è pronto.

Quest'ultimo approccio è stato adottato nel mio progetto, anche se la Epoll è bloccante indefinitamente sia nel Server che nel Client ci sono strutture come socket e coda messaggi che provocano sempre uno degli eventi EPOLLIN o EPOLLOUT per loro stessa natura, per cui la Epoll non si bloccherà mai indefinitamente ed i cicli while continueranno fin quando gli sarà necessario.

```
 // ciclo di attesa per i client
while (1)
{
    // Ogni 2 si generano nuove posizioni per i meteoriti
    sleep(2);
    generateMeteorites(meteoritesBuffer, 20);
    printf("[INFO] Nuove posizioni per i meteoriti generate: %s\n",
    ↪  meteoritesBuffer);

    // Attesa degli eventi di I/O
    struct epoll_event fdEvents[10];
    int triggeredEvents = waitForEvents(fdEvents, 10);

    for (int i = 0; i < triggeredEvents; i++)
```

2.2 Client: implementazione

L'implementazione del client è come segue:

- Controllo argomenti.
- Preparazione del socket ed opzioni del socket.
- Preparazione della struttura di indirizzo del server.
- Richiesta porta a cui inviare la richiesta di datagrammi.
- Registrazione dell'accoppiamento indirizzo Server <-> Client con connect così da poter mandare i propri datagrammi solo ed esclusivamente a quell'indirizzo ed associare a livello kernel l'indirizzo di destinazione e provenienza di dati.
- Controllo se c'è effettivamente un servizio in ascolto, controllo ICMP con send.
- Creazione Epoll.
- Creazione di due code messaggi POSIX: Client->Gioco , Gioco -> Client.
- Creazione di buffer per i meteoriti e messaggi al gioco.
- Invio messaggio di inizio gioco al client per invio meteore.
- Spawn del gioco in un altro terminale attraverso system.
- Variabili per il controllo del server(heartbeat).

- Inizio monitoraggio dei file descriptor del socket e code.
- Ciclo while infitio e gestione eventi.

Come si evince anche il client utilizza la Epoll come il server per monitorare il proprio socket, in aggiunta però vengono monitorate anche due code POSIX di messaggi per IPC fra il gioco e il client, questo per far si che il client e game siano isolati l'uno dall'altro.

Dopo la canonica creazione e set up del socket, ci sono due passaggi che normalmente non avvengono nei client basati puramente su UDP; per rendere più affidabile la nostra "non-connessione" al server associamo il nostro indirizzo del socket all'indirizzo del server tramite una *connect*, la connect non genera la connessione come normalmente farebbe ma permette al kernel di sapere che quel socket risponde e riceve sono a un determinato host e che qualunque altro host che provi a contattarlo verrà ignorato.

```
    extern void customConnection_init(int socket, struct sockaddr_in *serverAddr)
{
    // Argomenti della connect con cast
    struct sockaddr *internal = (struct sockaddr *)serverAddr;
    socklen_t size = sizeof(*serverAddr);

    if (connect(socket, internal, size) == -1)
    {
        customErrorPrinting("[ERROR] connect(): errore nella funzione connect\n");
        exit(EXIT_FAILURE);
    }
    else
    {
        fprintf(stdout, "[INFO] Indirizzo server registrato.\n");
    }
}
```

Questo ci permette poi in un secondo momento di catturare l'errore asincrono ICMP per un server che smette di comunicare data-grammi (che normalmente per un servizio UDP non sarebbe un problema) o un server che non è mai esistito.

Un volta associato il nostro indirizzo nel kernel utilizzeremo non *sendTo* e *recvFrom* ma le funzioni *send* e *recv*.

```
    extern int customSend(int socket, struct sockaddr_in *server, char *buffer)
{
    // Calcolo della dimensione del buffer + 1 per il terminatore
    // perché strlen non conta il terminatore.
    int size = strlen(buffer) + 1;

    // Invio del messaggio
    int sent = send(socket, buffer, size, 0);
    // Controllo dell'invio
    if (sent == -1)
    {
```

```
        customErrorPrinting("[ERROR] send(): la funzione ha generato un
        ↪  errore.\n");
        return 1;
    }
    else
    {
        // Controllo degli errori ICMP asincroni che non sono visti da send
        if (detectICMP(socket, server) == 1)
        {
            return 1;
        }
        fprintf(stdout, "[INFO] Inviati: %d byte\n", sent);
        return 0;
    }
}
```

Per i nostri scopi solo la send ha al suo interno il controllo ICMP, questo principalmente perché sarà usata per mandare continuamente ping al server ed all'inizio dell'avvio del client per farsi mettere in coda dal server.

I ping vengono realizzati ad inizio ciclo while, ogni 30 secondi, viene inviato un messaggio se il server è "DEAD" l'errore viene catturato e gestito e il client viene terminato assieme al gioco mediante messaggio alla coda e uscita dal ciclo while.

```
    // Valore temporale per generare un ping al server ogni 30 secondi
    double generatePingAfter = 30;
    time_t currentTimeElapsed;
    time_t timeFromLastPing;
    time(&timeFromLastPing);

    // Ciclo di gioco
    while (gameTerminated == 0)
    {
        // Ping al server per controllare la connessione ogni 30 secondi
        time(&currentTimeElapsed);
        if (difftime(currentTimeElapsed, timeFromLastPing) >= generatePingAfter)
        {
            int pingRes = customSend(clientSocket, &serverAddr, "PING");
            if (pingRes == 1)
            {
                customErrorPrinting("[ERROR] Il server non è raggiungibile\n");
                // Chiusura del gioco
                gameTerminated = 1;
                int s = sendMessageToQueue(queFd, "DEAD");
                printf("[INFO] Chiusura del gioco\n");
                continue;
            }
            time(&timeFromLastPing);
```

```
        }
        else
        {
            currentTimeElapsed = 0;
```

Per comunicare con il gioco il client utilizza due code, le code POSIX sono file che vengono creati all'interno del percorso */dev/mqueue* i file sono tenuti dal kernel sotto controllo e le informazioni al loro interno servono per gestire i messaggi, la priorità di essi ed evitare che chi scrive o legge possa provocare "race conditions" con opportuna sincronizzazione.

Vi sono delle limitazioni imposte dall'API delle code, innanzitutto le code hanno dei nomi specifici e che hanno una specifica sintassi : *"/nomecode"*, inoltre vi sono dei limiti al numero di messaggi per coda, alla grandezza dei messaggi che non possono eccedere ogni volta ed inoltre i processi hanno dei limiti imposti dal kernel sulla quantità di file descriptor aperti a loro nome per cui le code a fine uso vanno non semplicemente chiuse ma eliminate; è capitato più di una volta di saturare questo limite incorrendo nell'errore corrispondente, dovuto a molteplici chiusure anomale dei terminali che detenevano le code senza distruggerle.

Onde evitare problemi le code in sede di creazione le code sono create con flag specifici per impedirne la sovrascrittura.

```
    extern mq_open_data createQueue(char *queueName)
{
    /*
        Creazione della coda di messaggi.
        Flag:
        - O_RDWR: apertura in lettura/scrittura
        - O_NONBLOCK: apertura in lettura non bloccante
        - O_CREAT: crea la coda se non esiste
        - O_EXCL: se la coda esiste già ritorna errore
        - 0666: permessi di lettura e scrittura per tutti
        - NULL: attributi di default della coda per il sistema
    */
    mq_open_data data; // Struttura per i dati della coda di messaggi
    data.nameOfTheQueue = queueName;
    data.fileDescriptor = -1;
    data.error = -1;

    mqd_t mq = mq_open(data.nameOfTheQueue, O_RDWR | O_NONBLOCK | O_CREAT |
    ↪  O_EXCL, 0666, NULL);
    if (mq == (mqd_t)-1)
    {
        // Errore nella creazione della coda di messaggi, già esistente
        if (errno == EEXIST)
        {
            // Ne modifico il nome
            printf("[ERROR] mq_open(): Coda %s già esistente\n",
            ↪  data.nameOfTheQueue);
            srand(time(NULL));
            ↪  // Inizializzazione del generatore di numeri casuali
```

```
            char *newQueueName = calloc(20, sizeof(char));
            ↪  // 20 caratteri per il nome della coda
            sprintf(newQueueName, "%s%d", data.nameOfTheQueue, rand() % 999);
            ↪  // Aggiunta di un numero casuale al nome
            data.nameOfTheQueue = newQueueName;
            ↪  // Aggiornamento del nome della coda
            mq = mq_open(data.nameOfTheQueue, O_RDWR | O_NONBLOCK | O_CREAT |
            ↪  O_EXCL, 0666, NULL); // Creazione della coda con il nuovo nome
            if (mq == (mqd_t)-1)
            {
                printf("[ERROR] mq_open(): Errore creazione della coda %s:
                ↪  \n%s\n", strerrorname_np(errno), strerrordesc_np(errno));
                return data;
            }
            else
            {
                data.error = 0;
                data.fileDescriptor = mq;
                return data;
            }
        }
        else
        {
            printf("[ERROR] mq_open(): Errore creazione della coda %s: \n%s\n",
            ↪  strerrorname_np(errno), strerrordesc_np(errno));
            return data;
        }
    }
    // Coda creata con successo
    data.error = 0;
    data.fileDescriptor = mq;
    return data;
}
```

Questa funzione viene invocata la prima volta che un client viene avviato per creare le code, una per leggere dal proprio lato ed una per scrivere verso l'altro processo. Se il nome è già esistente tenterà di dargli un suffisso extra numerico pseudo-casuale, nella sfortuna di generare un nome già presente su 1000 numeri la funzione terminerà ed il client chiuso.

Le code e loro operazioni di lettura e scrittura normalmente sono bloccanti, nel mio progetto le code sono state settate a operazioni non-bloccanti con la flag apposita, questo approccio ha avuto (inconsciamente in principio) come lato positivo di poter costantemente sbloccare la Epoll del ciclo del client, dato che ogni operazione su una coda risulterà sempre disponibile generando un evento EPOLLIN/EPOLLOUT le funzioni di scrittura e lettura ritornano subito anche se le code sono vuote generando un errore EAGAIN e EWOULDBLOCK che vengono gestiti a parte.

```
/*
    Funzione per la ricezione dei messaggi dalla coda di messaggi passata come
    ↪  parametro.
```

```
    @param mq: descrittore della coda di messaggi
    @return: il messaggio ricevuto, NULL in caso di errore, "EMPTY" in caso di
    ↪  coda piena.
 */
extern char *receiveMessageFromQueue(int mq)
{
    char *buffer = calloc(8192, sizeof(char));
    if (mq_receive(mq, buffer, 8192, NULL) == -1)
    {
        if (errno == EAGAIN)
        {
            // Coda vuota, il blocco del thread è evitato con la flag O_NONBLOCK
            customErrorPrinting("[ERROR] mq_receive(): Coda vuota\n");
            return "EMPTY";
        }
        customErrorPrinting("[ERROR] mq_receive(): Errore ricezione messaggio\n");
        return NULL;
    }
    return buffer;
```

Il valore di 8192 è il valore di default degli attributi della coda per la grandezza dei messaggi che va rispettato altrimenti la funzione può andare in errore.

2.2.1 ICMP catch

Per poter scoprire se il server a cui ci stiamo rivolgendo esiste e può servirci su una determinata porta, oppure semplicemente pingare un server da cui stiamo ascoltando datagrammi che ci servono, dobbiamo necessariamente catturare un errore asincrono ICMP che nei socket UDP il kernel non notifica in maniera tale da poter essere preso in considerazione dal socket e dalle funzioni per la gestione della trasmissione dati.

A questo problema ci siamo dovuti arrendere alla nostra ignoranza e ci siamo rivolti ai moderni LLM nella speranza di ricevere degli opportuni suggerimenti o indicazioni per affrontare il problema.

Dopo aver ricevuto una risposta in cui si spiegava come raccogliere i messaggi che il kernel genera in questi casi, si è letta la documentazione su man7.org delle seguenti aree e funzioni: ip(7), recv(2) e csmg(3) con particolare enfasi sulla flag MSG_ERRQUEUE la quale permette di raccogliere i messaggi di errore in una coda ed analizzarli.

```
#define SO_EE_ORIGIN_NONE    0
#define SO_EE_ORIGIN_LOCAL   1
#define SO_EE_ORIGIN_ICMP    2
#define SO_EE_ORIGIN_ICMP6   3

struct sock_extended_err
{
    uint32_t ee_errno;   /* Error number */
    uint8_t  ee_origin;  /* Where the error originated */
```

```
        uint8_t  ee_type;    /* Type */
        uint8_t  ee_code;    /* Code */
        uint8_t  ee_pad;     /* Padding */
        uint32_t ee_info;    /* Additional information */
        uint32_t ee_data;    /* Other data */
        /* More data may follow */
    };

    struct sockaddr *SO_EE_OFFENDER(struct sock_extended_err *);
```

Questa struttura presa direttamente dal manuale, rappresenta un messaggio di errore del socket generato durante il suo funzionamento assieme ad essa è fornito anche l'indirizzo dell' "OFFENDER" ossia chi genera l'errore. Per poter raccogliere questi errori si devono abilitare a livello del socket, durante la creazione, nel livello SOL_IP la flag SO_IP_RECVERR.

L'estrazione vera e propria si ottiene attraverso funzione specificate nella pagina manuale csmg(3), queste funzioni hanno bisogno di dati di supporto (ancillary data) proveniente dalle comunicazioni del socket e che senza la quali non si può estrarre l'errore.

La struttura di supporto che conterrà le informazione da trasferire alla struttura **sock_extended_err** è la quella contenuta nel messaggio che estrarremo dalla coda errori **csmghdr** questa struttura dati è la seguente:

```
struct cmsghdr {
  size_t cmsg_len;    /* Data byte count, including header
                         (type is socklen_t in POSIX) */
  int    cmsg_level;  /* Originating protocol */
  int    cmsg_type;   /* Protocol-specific type */
/* followed by
  unsigned char cmsg_data[]; */
};
```

Per ottenere tutti i messaggi di errore che sono stati generati da una singola operazione su un socket dobbiamo ottenere l'intera coda da smaltire in secondo momento con le funzioni csmg(3), tutta la coda dei messaggi viene salvata in una struttura apposita ottenuta chiamando la funzione recvmsg() sul socket da controllare fornendogli la flag MSG_ERRQUEUE, inoltre la struttura in questione ha bisogno di un altra struttura dati a sua volta per immagazzinare i dati ottenuti dai registri di memoria definita in iovec(3type), seguono le due strutture dati:

```
#include <sys/uio.h>

       struct iovec {
           void   *iov_base;  /* Starting address */
           size_t  iov_len;   /* Size of the memory pointed to by iov_base. */
       };

       ----------------------
#include <sys/socket.h>

       struct msghdr {
               void         *msg_name;       /* Optional address */
               socklen_t     msg_namelen;    /* Size of address */
               struct iovec *msg_iov;        /* Scatter/gather array */
               size_t        msg_iovlen;     /* # elements in msg_iov */
               void         *msg_control;    /* Ancillary data, see below */
               size_t        msg_controllen; /* Ancillary data buffer len */
               int           msg_flags;      /* Flags on received message */
       };
```

A questo punto impostate opportunamente queste due strutture per ricevere dati invochiamo la funzione per ricevere la coda errori:

```
int bytes = recvmsg(socketFileDesc, &msg, MSG_ERRQUEUE);
```

Se la funzione ha successo il passo successivo è estrarre la struttura che contiene le informazioni di errore sul livello di errore e tipologia che vanno nella struttura csmghdr tramite un ciclo for in cui le funzioni/macro : CMSG_FIRSTHDR() e CMSG_NXTHDR() ci permettono di estrarre uno ad uno i messaggi di errore con la funzione CSMG_DATA():

```
// Dal primo messaggio di errore al successivo finchè non tornano NULL
    for (cmsg = CMSG_FIRSTHDR(&msg); cmsg != NULL; cmsg = CMSG_NXTHDR(&msg,
    ↪  cmsg))
    {
        // Se il messaggio di errore è di tipo IP_RECVERR e il livello è
        ↪  SOL_IP
        if (cmsg->cmsg_level == SOL_IP && cmsg->cmsg_type == IP_RECVERR)
        {
            // Estraiamo il messaggio di errore ICMP con la funzione
            ↪  CMSG_DATA()
            sock_err = (struct sock_extended_err *)CMSG_DATA(cmsg);

            // Se l'origine dell'errore è un ICMP avremo un valore nel campo
            ↪  ee_origin pari a SO_EE_ORIGIN_ICMP(2)
            if (sock_err->ee_origin == SO_EE_ORIGIN_ICMP)
            {
                // Copiamo l'indirizzo che orgina il messaggio ICMP in una
                ↪  variabile di tipo sockaddr_in
```

```
                    // dalla struttura sock_extended_err
                    struct sockaddr_in *offenderAddress = (struct sockaddr_in
                    ↪  *)(sock_err + 1);
```

Ottenuto l'indirizzo dell' "OFFENDER" bisogna vedere che codice di errore ICMP abbiamo ricevuto, solo alcuni errori sono di nostro interesse, leggendo la documentazione sull'header di un pacchetto ICMP gli errori che possono indicare un problema con l'host a cui ci si sta rivolgendo sono del gruppo: 3 e valori di errore dallo 0 al 3:

```
if (sock_err->ee_type == 3)
    {
        if (sock_err->ee_code == 0 ||
            sock_err->ee_code == 1 ||
            sock_err->ee_code == 2 ||
            sock_err->ee_code == 3)
        {
            // Se l'indirizzo che ha originato l'errore è uguale a quello che
            ↪  abbiamo passato come parametro
            if (addressToCheckAgainst->sin_addr.s_addr ==
            ↪  offenderAddress->sin_addr.s_addr)
            {
                // Stampa del messaggio di errore
                printf("[ERROR] Errore ICMP rilevato:");
                // Ci sono errori che non hanno una stringa di errore associata o
                ↪  che la funzione associata non marca con errno
                if (strerror(sock_err->ee_errno) != NULL ||
                ↪  strerror(sock_err->ee_errno) != "Success")
                {
                    printf(" %s- ", strerror(sock_err->ee_errno));
                }
                else
                {
                    printf("Flag errno non settata dalla funzione o sconosciuta-
                    ↪  ");
                }

                printf("ICMP error: Tipo: %d, Codice: %d\n", sock_err->ee_type,
                ↪  sock_err->ee_code);
                return 1; // Restituzione del valore di errore
            }
        }
        else
        {
        // Se il codice dell'errore non è di nostro interesse
        return 0;
        }
```

2.3 Game engine: implementazione

Il motore di gioco si sviluppa in un proprio terminale lanciato attraverso il client con al chiamata di sistema system e una stringa che viene eseguita nel terminale appena evocato.

```
// Apertura del terminal con spawn di un processo figlio dedito al gioco
int systemResult = 0;
char *baseCommand = "gnome-terminal -- ./Game";
// Preparazione del comando per l'apertura del terminale con argomenti i nomi
↪  delle code
char *composedCommand = calloc(strlen(baseCommand) +
                                    strlen(queueName) +
                                    strlen(queueName_2) +
                                    2 + 1,
                               sizeof(char)); // 2 spazi e terminatore
sprintf(composedCommand, "%s %s %s", baseCommand, queueName, queueName_2);
systemResult = system(composedCommand);
if (systemResult == -1)
{
    customErrorPrinting("[ERROR] Apertura del terminale fallita\n");
    exit(EXIT_FAILURE);
}
```

Al proprio interno la funzione system utilizza altre chiamate di sistema come: fork(2), execl(3), e waitpid(2) ma a costo di risorse maggiori in cambio di semplicità e convenienza d'uso. A differenza di fork() non abbiamo modo di ottenere da system il pid del processo che viene generato, per questa ragione si sono implementate le code IPC per permettere ai processi game e client di parlarsi in isolamento l'uno dall'altro.

Il ciclo di vita del game engine è come segue:

- Creazione delle code messaggi per le comunicazione IPC
- Creazione istanza Epoll
- Preparazione del terminale per il gioco
- Ciclo infitio di gioco con monitor eventi Epoll ed eventi di gioco

La politica che gestisce il motore di gioco si basa sul principio di aggiornare costantemente il campo da gioco ad ogni evento, ridisegnando l'intera grigla dopo aver internamente aggiornato i parametri necessari, questo approccio dà l'illusione a chi sta davanti al PC di eventi in tempo reale quando in realtà ciò che si vede è una lunga sequenza di "slide" che rappresentano lo stato interno di parametri del gioco ad un determinato momento temporale.

```
#define ROWS 11    // 10 righe di gioco + 1 riga di avviso
#define COLUMNS 20 // 20 colonne di gioco

#define EMPTY_CELL '.'
#define VOID_CELL ' '
```

```c
#define METEORITE_CELL 'o'
#define SHIP_CELL '^'
#define EXPLOSION_CELL 'X'
#define DANGER_MESSAGE "METEORITE IN ARRIVO!"
#define DANGER_RANGE 3
#define LEFT_ARROW "\x1b[D"
#define RIGHT_ARROW "\x1b[C"

// Griglia di gioco 10*20
char **grid;
/*
    Posizione della navicella che si può muovere
    Solo in una riga, aggiornato a ogni iterazione
*/
int shipPosition = 9; // Posizione iniziale e centrale
// Archivio meteoriti 11*20
char **meteoritesBuffer;
int dangerMessageOn = 0;
```

Queste sono variabili globali del file che contiene il sistema di disegno della griglia, composta da 10 righe più una di servizio e venti colonne, viene salvata inizialmente nell'array multi dimensionale apposito e da li in poi gestito e modificato a seconda degli eventi.

L'input dei giocatori è consentito solo con le frecce direzionali, e solo a destra e sinistra. Per cui la Epoll del motore del gioco prende in carico il file descriptor dello standard input e ne monitora gli eventi, data la capacità di Epoll di sopportare un gran numero di eventi contemporanei non si rischia di compromettere gli altri componenti rovinando l'esperienza di gioco.

```c
else if (events[i].data.fd == STDIN_FILENO && events[i].events & EPOLLIN) //
↪  Controllo se l'evento è relativo all'input da tastiera
                {
                    // Lettura da tastiera
                    char c[10];
                    int n = read(STDIN_FILENO, c, 9);
                    quittingGame = checkTheInput(c, n);
                    if (quittingGame == 1)
                    {
                        gameOver = 1;
                        sleep(2);           // Attesa per la visualizzazione del
                        ↪  game over
                        setTerminalMode(0); // Modalità normale del terminale
                        // Pulizia della griglia di gioco
                        printf("\033[H\033[J");
                        printf("[INFO] Uscita dal gioco in corso\n");
                        sleep(2);
                        break; // Esco dal ciclo for e torno al while
                    }
```

```
}
```

La Epoll cattura gli input del file descriptor e li passa ad un funzione apposita che controlla di che input si tratti, scartando quelli che non servono, ed agisce poi internamente al sistema di disegno modificando la griglia di gioco per poi stamparla in un secondo momento, questa operazione avviene così velocemente che per la persona davanti al terminale sembra che avvenga tutto in tempo reale.

```
extern int checkTheInput(char *input, int inputSize)
{
    // Gli input essendo tutti in raw mode non hanno null terminator
    // Per fare quindi dei controlli con stringhe predefinite, si
    // deve aggiungere il null terminator manualmente.
    input[inputSize] = '\0';

    // Controllo tipo di input ricevuto
    if (strcmp(input, LEFT_ARROW) == 0)
    {
        shipMovement(-1);
    }
    else if (strcmp(input, RIGHT_ARROW) == 0)
    {
        shipMovement(1);
    }
    else if (strcmp(input, "q") == 0)
    {
        gameOver();
        setTerminalMode(0);
        printf("Uscita dal gioco a breve.\n");
        return 1; // Chiusura del gioco
    }
    return 0; // Nessuna chiusura del gioco
}
```

Qualora l'utente volesse uscire dal gioco prima del suo naturale completamento, anche perché non è stata prevista alcuna altra modalità di uscita come un punteggio limite o altro, può premere il tasto Q e segnalare al gioco l'uscita.

Gli asteroidi vengono generati in delle posizioni casuali da parte del server, su una stringa di 20 caratteri numerici di 1 e 0 solo 5 posizioni casualmente scelte avranno il valore 1. Il client riceve questa stringa e la passa al motore di gioco tramite coda messaggi, il motore di gioco a sua volta prende in carico la stringa e la processa come segue:

```
extern void addMeteors(char *newRowBuffer)
{
    // Per dare la possibilità ai giocatori di evitare i meteoriti è necessario
```

```
// dare uno "spazio" di manovra, quindi prima di inserire i meteoriti nella
↪  griglia
// si genera un numero casuale tra 0 e 5 per decidere se inserire o meno i
↪  meteoriti
srand(time(NULL));
int insertMeteorites = rand() % 5 + 1;

// Aggiungo a prescidere i meteoriti nell'archivio se c'è spazio
for (int i = 0; i < 10; i++)
{
    if (meteoritesBuffer[i][0] == 'E')
    {
        meteoritesBuffer[i][0] = 'F'; // Flag per la riga piena, FULL
        for (int j = 1; j < 21; j++)
        {
            // J-1 perché il primo carattere è il flag per l'array di archivio
            ↪  ma non per il buffer in arrivo
            // che è di 20 caratteri
            meteoritesBuffer[i][j] = newRowBuffer[j - 1];
        }
        break;
    }
}

// I meteoriti sono rappresentati da 'o' nella griglia
// E sono inseriti sempre a partire dalla griglia 0
// Se il numero casuale è minore di 3, non vengono inseriti meteoriti
// Se il numero casuale è maggiore  3, vengono inseriti i meteoriti
// Dall'archivio meteoriti
if (insertMeteorites > 3)
{
    int found = 0;
    while (found == 0)
    {
        // Scorro fino a trovare una F
        for (int i = 0; i < 10; i++)
        {
            if (meteoritesBuffer[i][0] == 'F')
            {
                // Inserisco i meteoriti nella griglia
                for (int j = 1; j < 21; j++)
                {
                    // J-1 perché la griglia è di 20 colonne non 21
                    if (meteoritesBuffer[i][j] == '1')
                    {
                        grid[0][j - 1] = METEORITE_CELL;
                    }
                    else
```

```
                        {
                            grid[0][j - 1] = EMPTY_CELL;
                        }
                        // Al contempo svuoto l'archivio meteoriti per la riga
                        meteoritesBuffer[i][j] = '0';
                    }
                    // Flag per la riga vuota, EMPTY
                    meteoritesBuffer[i][0] = 'E';
                    found = 1; // Meteoriti inseriti, esci dal while e for
                    break;
                }
            }
        }
    }
}
```

Il motore di gioco ha a sua volta un archivio di meteore da inserire nel campo di gioco, dato che il client sostanzialmente ogni due secondi gli passa un elenco ricevuto dal server, ma potenzialmente anche di più.

Mantiene un archivio di queste meteore in array di 20 più 1 colonna di controllo, la colonna di controllo può avere valori 'E' o 'F' che rappresentano una riga di meteore vuota o piena, ed in base a questo il gioco archivia le nuove meteore in uno slot vuoto o ne prende una da uno slot pieno(E lo svuota). L'aggiunta di meteore è casuale nel senso che le posizioni sono prese ed inserite in base al valore di un numero generato casualmente con la rand, questo per dare "spazio" al giocatore che altrimenti si ritroverebbe letteralmente un muro di meteore che in certe combinazioni non sarebbero evitabili portando al game over certo.

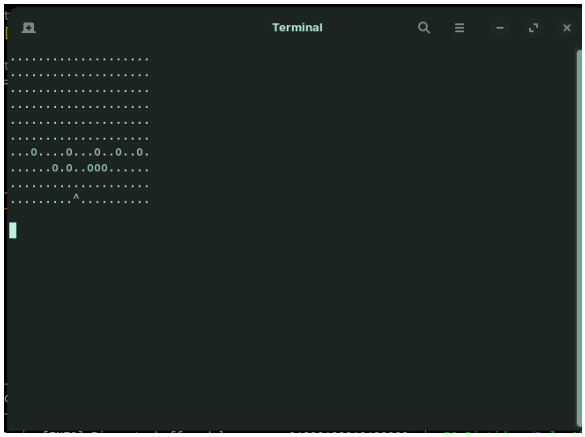

Figure 4: Griglia di gioco con meteore

L'aggiornamento della griglia di gioco avviene in concomitanza di eventi sulla Epoll, quando la coda messaggi è pronta in lettura la Epoll prova a leggere se ci sono messaggi, in concomitanza con questo evento se ci sono nuovi messaggi vengono aggiunte le meteore al motore di gioco e la griglia viene aggiornata, al contrario l'esecuzione prevede altri casi o prosegue senza fare nulla.

```
fromClientBuffer = receiveMessageFromQueue(queFd);
if (fromClientBuffer == NULL || strcmp(fromClientBuffer, "DEAD") == 0)
{
    setTerminalMode(0);      // Rimetto il terminale in modalità normale
    printf("\033[H\033[J"); // Pulizia del terminale
    printf("[ERROR] Server assente o problema con la coda messaggi.\n");
    gameOver = 1;
    quittingGame = 1;
    break; // Esco dal ciclo for e torno al while
}
else if (strcmp(fromClientBuffer, "EMPTY") == 0)
{
    // Coda vuota non faccio nulla e faccio andare avanti il gioco
    continue;
}
else
{
    // Aggiunta dei meteoriti alla griglia di gioco
    addMeteors(fromClientBuffer);
    // Ridisegno la griglia di gioco ad ogni aggiunta di meteoriti
    gameOver = redrawRowsTicker();
    if (gameOver == 1)
    {
        // Il gioco è finito
        quittingGame = 1;
        break; // Esco dal ciclo for e torna al while
```

La funzione di redraw aggiorna tutte le posizioni delle meteore nella griglia a partire dalla riga 1 fino alla riga prima della nave. Questo approccio dà l'illusione che le meteore effettivamente scendano verso il giocatore, il meccanismo dietro è semplicemente la copia cella per cella della riga immediatamente sopra.

Durante il ciclo di aggiornamento vengono controllate anche la distanza fra nave e un meteorite se si trova fino a 3 celle sopra di essa, verrà mostrato il messaggio di avviso sulla riga di servizio. Infine controlla se facendo il prossimo aggiornamento delle righe la nave si trova esattamente in una posizione in cui collide con un meteora sopra di essa.

```
extern int redrawRowsTicker()
{
    // Controllo per l'avviso di pericolo
    checkDanger();
    // Controllo per la collisione tra nave e meteorite
    if (checkCollision() == 1)
    {
        // Se c'è una collisione, si attiva il game over
        for (int i = 0; i < COLUMNS; i++)
        {
```

```
            grid[ROWS - 2][i] = EXPLOSION_CELL;
        }
        // Testo di game over nella riga di avviso
        char *gameOverBuff = "GAME OVER";
        size_t len = strlen(gameOverBuff);
        for (int i = 0; i < COLUMNS; i++)
        {
            grid[ROWS - 1][i] = VOID_CELL;
        }
        for (int i = 0; i < len; i++)
        {
            grid[ROWS - 1][i] = gameOverBuff[i];
        }
        // Stampa della griglia di gioco
        printGrid();
        sleep(3);   // Attesa per la visualizzazione griglia di gioco
        gameOver(); // Funzione per il game over
        setTerminalMode(0);
        printf("Game over, uscita dal gioco a breve.\n");
        sleep(3); // Attesa per la visualizzazione del game over
        return 1; // Game over
    }
    // Avanzamento della griglia di gioco
    // Dalla righa 8 alla riga 1
    int nextRowArray = 8;
    while (nextRowArray > 0)
    {
        for (int i = 1; i < COLUMNS; i++)
        {
            grid[nextRowArray][i] = grid[nextRowArray - 1][i];
        }
        nextRowArray--;
    }
    // La riga 0 diventa vuota
    for (int i = 0; i < COLUMNS; i++)
    {
        grid[0][i] = EMPTY_CELL;
    }
    // Stampa della griglia di gioco
    printGrid();

    return 0; // Nessun game over
}
```

2.3.1 Terminal cooked e raw

Una delle problematica che ci si è posti mentre si pensava a come realizzare un animazione fluida del gioco è di come impostare il terminale per catturare continuamente l'input dei giocatori senza

richiedere la pressione del tasto ENTER e senza mostra il testo classico di path del terminale.

Dopo varie ricerche online e varie domande agli LLM e test, si è optato per impostare il terminale in modalità *"raw"*, questa modalità risponde alle nostre esigenze implementative, le due modalità del terminale sono così definite da GNU:

- **COOKED MODE:***A terminal port in cooked mode provides some standard processing to make the terminal easy to communicate with. For example, under unix, cooked mode on input reads from the terminal a line at a time and provides editing within the line, while cooked mode on output might translate linefeeds to carriage-return/linefeed pairs.*
- **RAW MODE:***A terminal port in raw mode disables all of that processing. In raw mode, characters are directly read from and written to the device without any translation or interpretation by the operating system. On input, characters are available as soon as they are typed, and are not echoed on the terminal by the operating system.*

Per rendere il terminale in modalità raw si è scelto di utilizzare la libreria *termios.h* disponibile in C, attraverso di essa si possono recuperare le impostazioni del terminale e modificarle per i propri scopi, non conoscendo nel dettaglio le impostazioni del terminale e dei vari livelli di opzioni si è proceduto per tentativi leggendo la documentazione su man7 per termios(3) e con un aiuto dai LLM si è raggiunto l'obbiettivo prefissato.

```c
/*
    Manipolazione terminale per la lettura di un singolo carattere
    @param mode: modalità di operazione del terminale[0,1]
    - 0: modalità normale (cooked)
    - 1: modalità game mode(raw)
*/
extern void setTerminalMode(int mode)
{
    // Struttura per le impostazioni del terminale
    struct termios termios_p;

    // Impostazioni del terminale corrente
    tcgetattr(STDIN_FILENO, &termios_p);

    if (mode == 0)
    {
        // Modalità normale (cooked)
        termios_p.c_lflag |= (ICANON | ECHO); // Flag per modalità canonica e echo
    }
    else if (mode == 1)
    {
        // Modalità game mode (raw)
        termios_p.c_lflag &= ~(ICANON | ECHO); // Disabilitazione di modalità
        ↪  canonica e echo
        termios_p.c_cc[VMIN] = 1;              // Numero di caratteri minimi per
        ↪  attivare la lettura
        termios_p.c_cc[VTIME] = 0;             // Timeout per la modalità non
        ↪  canonica
    }

    // Applicazione delle impostazioni
    tcsetattr(STDIN_FILENO, TCSANOW, &termios_p);
}
```

Mediante operatori bitwise OR, AND e NOT si modificano i bit delle flag della struttura che contiene le opzioni del terminale contenute nel campo *c_lflag* rimuovendo la flag per la modalità canonica e l'echo; gli altri due campi sono utilizzati per specificare le opzioni temporali per la funzione read che legge i caratteri dal terminale, leggendo la documentazione delle libreria per la modalità non canonica(raw) ci sono 4 casi d'uso, il caso che interessa i nostri scopi progettuali è il seguente tratto direttamente dalla documentazione su man7:

- **MIN $>$ 0, TIME $==$ 0 (blocking read):** *read(2) blocks until MIN bytes are available, and returns up to the number of bytes requested.*

Rispettivamente *c_cc[VMIN]* rappresenta il valore MIN di caratteri affinché la read possa tornare il numero di byte letti, *cc_[VTIME]* è il tempo di timeout della funzione read del terminale che se impostato a zero la rende bloccante ergo il mio terminale accetterà solo 1 carattere alla volta

e tornerà subito il valore al processo controllante, questo mi permette di gestire l'input di gioco come singole pressioni di tasti senza che appaiono allo schermo (echo) ed anche qualora si tenga premuto il pulsante l'input sarà "frazionato" singolarmente; grazie all' Epoll il flooding di input non blocca/rallenta gli altri componenti del gioco.

3 Installazione ed avvio

Il gioco viene compilato tramite toolchain CMAKE, il file di cmake è strutturato come segue:

```
    cmake_minimum_required(VERSION 3.10)

# Project name
project(GameServerClient)

# Set the C standard
set(CMAKE_C_STANDARD 99)

# Include directories
include_directories(${CMAKE_SOURCE_DIR}/../lib)

# Source files
set(SERVER_SOURCES ${CMAKE_SOURCE_DIR}/../Server/Server.c)
set(CLIENT_SOURCES ${CMAKE_SOURCE_DIR}/../Client/Client.c)
set(GAME_SOURCES    ${CMAKE_SOURCE_DIR}/../gameEngine.c)

# Library files
set(LIB_SOURCES
    ${CMAKE_SOURCE_DIR}/../lib/wrappers/customRecvFrom.c
    ${CMAKE_SOURCE_DIR}/../lib/wrappers/basicWrappers.c
    ${CMAKE_SOURCE_DIR}/../lib/wrappers/pollUtils.c
    ${CMAKE_SOURCE_DIR}/../lib/wrappers/customSendTo.c
    ${CMAKE_SOURCE_DIR}/../lib/wrappers/customErrorPrinting.c
    ${CMAKE_SOURCE_DIR}/../lib/wrappers/addressTools.c
    ${CMAKE_SOURCE_DIR}/../lib/argChecker.c
    ${CMAKE_SOURCE_DIR}/../lib/wrappers/customRecvFrom.h
    ${CMAKE_SOURCE_DIR}/../lib/wrappers/basicWrappers.h
    ${CMAKE_SOURCE_DIR}/../lib/wrappers/pollUtils.h
    ${CMAKE_SOURCE_DIR}/../lib/wrappers/customSendTo.h
    ${CMAKE_SOURCE_DIR}/../lib/wrappers/customErrorPrinting.h
    ${CMAKE_SOURCE_DIR}/../lib/wrappers/addressTools.h
    ${CMAKE_SOURCE_DIR}/../lib/wrappers/bufHandlers.c
    ${CMAKE_SOURCE_DIR}/../lib/wrappers/bufHandlers.h
    ${CMAKE_SOURCE_DIR}/../lib/wrappers/customConnection.c
    ${CMAKE_SOURCE_DIR}/../lib/wrappers/customConnection.h
    ${CMAKE_SOURCE_DIR}/../lib/wrappers/customUDPTransmission.c
    ${CMAKE_SOURCE_DIR}/../lib/wrappers/customUDPTransmission.h
    ${CMAKE_SOURCE_DIR}/../lib/gamelogics/meteorites.c
    ${CMAKE_SOURCE_DIR}/../lib/gamelogics/meteorites.h
```

```
    ${CMAKE_SOURCE_DIR}/../lib/gamelogics/drawingField.h
    ${CMAKE_SOURCE_DIR}/../lib/gamelogics/drawingField.c
    ${CMAKE_SOURCE_DIR}/../lib/wrappers/customICMP.c
    ${CMAKE_SOURCE_DIR}/../lib/wrappers/customICMP.h
    ${CMAKE_SOURCE_DIR}/../lib/wrappers/customQueIPC.h
    ${CMAKE_SOURCE_DIR}/../lib/wrappers/customQueIPC.c
    ${CMAKE_SOURCE_DIR}/../lib/argChecker.h
)

# Add executable targets
add_executable(Server ${SERVER_SOURCES} ${LIB_SOURCES})
add_executable(Client ${CLIENT_SOURCES} ${LIB_SOURCES})
add_executable(Game ${GAME_SOURCES} ${LIB_SOURCES})
```

Ogni singolo eseguibile è compilato assieme all'intera libreria di file sorgente ed header per le varie funzioni necessarie al funzionamento, anche se alcune funzioni non sono utilizzate, questa scelta oltre che di convenienza puramente organizzativa ha anche alcuni vantaggi che derivano dalla logica implementativa di alcune funzioni della libreria: funzioni come la Epoll in *PollUtils.c* mantiene al suo interno il file descriptor dell'istanza di Epoll, dato che tutti e tre gli eseguibili gestiscono un monitor di Epoll non vogliamo che tutti condividano questo file descriptor. Per cui ogni eseguibile manterrà la propria copia della libreria *lib* ed i valori a loro associati.

Nella cartella */Builds* della repository c'è il file CMAKE da avviare via riga di comando:

```
luc1@luc1-B650M-PG-Riptide:~/Desktop/Esame-Pratico-Reti-19-12-2024/Builds$
↪  cmake .
-- Configuring done
-- Generating done
-- Build files have been written to:
↪  /home/luc1/Desktop/Esame-Pratico-Reti-19-12-2024/Builds
```

Dopodichè lanciando il comando *make* i file eseguibili vengono generati:

```
luc1@luc1-B650M-PG-Riptide:~/Desktop/Esame-Pratico-Reti-19-12-2024/Builds$ make
Consolidate compiler generated dependencies of target Server
[  2%] Building C object
↪  CMakeFiles/Server.dir/home/luc1/Desktop/Esame-Pratico-Reti-19-12-2024/lib/wrappers/customUDPTra
[  4%] Linking C executable Server
[ 33%] Built target Server
Consolidate compiler generated dependencies of target Client
[ 35%] Building C object
↪  CMakeFiles/Client.dir/home/luc1/Desktop/Esame-Pratico-Reti-19-12-2024/Client/Client.c.o
[ 37%] Building C object
↪  CMakeFiles/Client.dir/home/luc1/Desktop/Esame-Pratico-Reti-19-12-2024/lib/wrappers/customUDPTra
[ 39%] Linking C executable Client
[ 66%] Built target Client
Consolidate compiler generated dependencies of target Game
```

```
[ 68%] Building C object
↪  CMakeFiles/Game.dir/home/luc1/Desktop/Esame-Pratico-Reti-19-12-2024/lib/wrappers/customUDPTrans
[ 70%] Linking C executable Game
[100%] Built target Game
```

A questo punto è possibile lanciare gli eseguibili per provare l'applicazione.

4 Conclusioni finali

Per quanto ci si è sforzati di assemblare un progetto almeno funzionante nei minimi minimi termini richiesti, ci sono molti aspetti che sono stati sorvolati o mal implementati; la gestione della memoria è la prima vittima di questo progetto, più di una esecuzione è andata in segfault e valgrind ha aiutato a mettere un pezzo di scotch su più di una falla della nave, ma nonostante ciò la gestione della memoria rimane pessima ma sopportabile per non far andare interamente in crash l'applicazione.

Altra mancanza è un test dell'applicazione su un effettiva connessione remota Server Client, non si è riusciti a provare tale scenario rimanendo confinati in LOCALHOST, questo è un grosso limite al progetto e me ne dispiaccio di non averlo neanche scalfito, sarà probabilmente oggetto di discussione in sede d'esame.

In conclusione si spera che il materiale che si è presentato soddisfi almeno i minimi requisiti della traccia proposta, e che sia accettabile in sede di giudizio, rimane comunque un assembramento di funzionalità prone all'errore al minimo cambiamento della codebase e difficilmente mantenibile.