Relazione Progetto di Reti di Calcolatori - StarShip

Nome: Vincenzo
Cognome: Franchetti
Matricola: 0124002616

Indice

---

1. Introduzione e Descrizione del Progetto

Questo progetto realizza un'applicazione client-server in Python basata sul protocollo **UDP** (User Datagram Protocol), simulando la comunicazione in tempo reale tra una **navicella spaziale** (client) e un **ambiente popolato da meteoriti** (gestito dal server). La scelta di UDP come protocollo di comunicazione permette uno scambio rapido e continuo di dati tra le due entità, senza la necessità di una connessione stabile e permanente, rendendo UDP ideale per questo tipo di simulazioni dove la velocità è prioritaria rispetto alla garanzia di consegna dei pacchetti.

Struttura del Progetto

Il progetto è strutturato in due componenti principali, ognuno dei quali svolge compiti ben definiti, contribuendo alla simulazione dell'ambiente in modo coordinato:

- **Client (navicella)**: Il client rappresenta la navicella spaziale e svolge tre ruoli fondamentali:

    1. **Interfaccia utente**: attraverso l'uso di una GUI creata con Tkinter, consente all'utente di interagire con il gioco tramite comandi di movimento (ad esempio, spostamenti a destra, sinistra, su o giù).
    2. **Invio di comandi**: il client rileva i comandi dell'utente e li converte in pacchetti UDP, inviandoli al server in tempo reale per un'elaborazione rapida.

3. **Aggiornamento grafico**: riceve i dati di stato aggiornati dal server (ad esempio, posizione dei meteoriti) e utilizza queste informazioni per aggiornare costantemente la grafica, mantenendo un'immagine chiara e coerente della situazione dell'ambiente.

- **Server (meteoriti)**: Il server gestisce l'ambiente e la logica associata agli ostacoli, simulati dai meteoriti. Anche qui, le sue responsabilità sono diverse:

  1. **Ricezione dei comandi**: il server ascolta i pacchetti inviati dal client e interpreta i comandi ricevuti.
  2. **Elaborazione della situazione**: aggiorna la posizione della navicella in relazione ai meteoriti, valutando eventuali collisioni o movimenti critici che possono influire sull'interazione del gioco.
  3. **Risposta al client**: dopo aver calcolato lo stato aggiornato dell'ambiente, invia i dati al client, permettendogli di mantenere una rappresentazione costantemente aggiornata dell'ambiente.

Questa divisione di ruoli e compiti tra client e server permette di separare la logica di controllo utente (gestita dal client) dalla logica di ambiente e ostacoli (gestita dal server), facilitando l'espansione o la modifica di ciascuna componente in futuro.

Moduli Utilizzati

Il progetto sfrutta vari moduli Python che sono fondamentali per gestire la comunicazione di rete, l'interfaccia grafica e la gestione delle immagini:

- **socket**: Il modulo `socket` permette di creare e gestire la comunicazione in rete tra client e server. Nello specifico, vengono creati socket UDP (`AF_INET` e `SOCK_DGRAM`) per inviare e ricevere pacchetti in modo rapido e senza connessione. La funzione `socket.sendto` permette di inviare pacchetti con i comandi del client o le risposte del server, mentre `socket.recvfrom` gestisce la ricezione dei pacchetti, consentendo una comunicazione bidirezionale.

- **Tkinter**: `Tkinter` è utilizzato per gestire l'interfaccia grafica del client, creando una finestra di gioco dove viene visualizzata la navicella, l'ambiente e i meteoriti. Tkinter permette di disegnare e aggiornare gli elementi grafici in tempo reale in base ai dati ricevuti, assicurando una reattività immediata ai comandi utente. È impiegato per gestire eventi di input (come la pressione di tasti) e aggiornamenti visivi dinamici, che rappresentano i movimenti della navicella e i cambiamenti di posizione dei meteoriti.

- **Pillow (PIL)**: `Pillow` è una libreria di gestione delle immagini che permette di caricare e mostrare immagini della navicella e dei meteoriti. La libreria consente di manipolare facilmente le immagini, ridimensionarle o trasformarle per visualizzare in

modo corretto gli oggetti grafici nella GUI. In questo modo, il progetto raggiunge una rappresentazione visiva semplice ma chiara e accattivante, che aiuta a rendere l'esperienza di gioco più coinvolgente.

Questa combinazione di moduli consente una comunicazione fluida e continua tra client e server e garantisce una rappresentazione grafica funzionale, creando una simulazione interattiva e reattiva.

2. Componenti del Codice

In questa sezione vengono illustrate le principali funzioni e operazioni che costituiscono la base del progetto, sia per il client (`Client_navicella.py`) sia per il server (`Server_meteoriti.py`). Ogni funzione sarà spiegata in dettaglio, analizzando il suo scopo, il funzionamento e il ruolo specifico all'interno dell'applicazione. Le funzioni del client gestiscono la logica di interfaccia e la comunicazione verso il server, mentre quelle del server si occupano della gestione dell'ambiente e del ricevimento delle informazioni dalla navicella.

`ricevi_meteoriti`

```
def ricevi_meteoriti(socket_client, posizione_navicella, canvas, meteorite_img, info_label):
    """
     Funzione che riceve la posizione dei meteoriti dal server e aggiorna la griglia.
    """
    global meteoriti_posizioni, game_over
     client_running = True  # Variabile per mantenere il ciclo di ricezione attivo
    while client_running:
        try:
            # Ricezione dati dal server
            dati, _ = socket_client.recvfrom(1024)
            message = dati.decode()

            if message == 'clear':
                    # Se il server invia 'clear', rimuove tutti i meteoriti dalla griglia
                canvas.delete('meteorite')
                meteoriti_posizioni.clear()
            elif ',' in message:
                # Gestione della posizione del meteorite
                            posizione_meteorite = tuple(map(int, message.split(',')))
```

```
                        meteoriti_posizioni[posizione_meteorite] =
posizione_meteorite
                    disegna_meteorite(canvas, posizione_meteorite,
meteorite_img)

                # Controllo collisione tra navicella e meteorite
                                    if  posizione_meteorite  ==
tuple(posizione_navicella):
                    game_over = True
                    client_running = False
                     canvas.create_text(150, 150, text="GAME OVER",
font=("Helvetica", 30), fill="red")
                    return
            else:
                 print(message)  # Debug per messaggi di benvenuto
o altri messaggi del server
            aggiorna_info(info_label)
        except SyntaxError as e:
              print(f"Errore di sintassi durante la ricezione dei
dati: {e}")
        except socket.error as e:
            print(f"Errore durante la ricezione dei dati: {e}")
        except ValueError as e:
            print(f"Errore nel parsing dei dati ricevuti: {e}")
        except Exception as e:
            print(f"Errore sconosciuto: {e}")
```

Descrizione della Funzione

La funzione `ricevi_meteoriti` è progettata per ricevere i dati relativi alla posizione dei meteoriti dal server e aggiornare graficamente la posizione degli ostacoli sul canvas dell'interfaccia. La funzione esegue un ciclo continuo per garantire la ricezione in tempo reale delle informazioni inviate dal server.

- **Parametri**:
  - `socket_client`: rappresenta il socket attraverso cui il client riceve i dati dal server.
  - `posizione_navicella`: una lista o una tupla che contiene la posizione attuale della navicella.
  - `canvas`: oggetto di Tkinter che visualizza graficamente la navicella e i meteoriti.
  - `meteorite_img`: immagine del meteorite che viene disegnata sul canvas.
  - `info_label`: etichetta di Tkinter usata per visualizzare le informazioni di gioco.

Funzionamento Dettagliato

- **Inizializzazione della Variabile di Controllo**:
  - `client_running` è una variabile booleana usata per mantenere il ciclo attivo finché la ricezione dei dati è necessaria.
- **Ricezione dei Dati**:
  - La funzione chiama `socket_client.recvfrom(1024)` per ricevere un pacchetto di dati dal server (fino a 1024 byte).
  - Il pacchetto viene decodificato in formato stringa e assegnato a `message`.
- **Elaborazione del Messaggio Ricevuto**:
  - **Messaggio 'clear'**: Se il server invia la stringa `"clear"`, la funzione interpreta questo comando come un segnale per rimuovere tutti i meteoriti attualmente presenti sul canvas. Questo è gestito con `canvas.delete('meteorite')`, e `meteoriti_posizioni.clear()` rimuove le posizioni dei meteoriti salvate.
  - **Posizione dei Meteoriti**: Se il messaggio contiene una virgola (e quindi rappresenta una posizione), viene convertito in una tupla di coordinate `(x, y)` per la posizione del meteorite. La funzione chiama `disegna_meteorite` per aggiornare graficamente il meteorite sul canvas.
  - **Controllo Collisioni**: Dopo aver disegnato il meteorite, la funzione verifica se la posizione del meteorite coincide con quella della navicella. In caso di collisione, imposta `game_over` su `True`, interrompe il ciclo e visualizza la scritta "GAME OVER" al centro del canvas.
- **Aggiornamento Informazioni di Gioco**:
  - Dopo ogni operazione, la funzione chiama `aggiorna_info(info_label)` per aggiornare le informazioni mostrate all'utente.
- **Gestione degli Errori**:
  - Vengono gestite varie eccezioni per evitare interruzioni in caso di errori, tra cui `SyntaxError`, `socket.error`, `ValueError`, e altre eccezioni generiche, ciascuna con un messaggio di errore specifico.

`invia_comando`

```
def invia_comando(comando, socket_client):
    """
    Invia un comando al server.
```

```
    """
    try:
        socket_client.sendto(comando.encode(), (HOST, PORT))
    except socket.error as e:
        print(f"Errore durante l'invio del comando: {e}")
```

Descrizione della Funzione

La funzione `invia_comando` si occupa di inviare un comando al server. Questo permette al client di trasmettere le istruzioni di controllo (come i comandi di movimento) attraverso il socket UDP, inviandole all'indirizzo specifico del server.

- **Parametri**:
  - `comando`: una stringa che rappresenta il comando da inviare al server (es. movimenti o azioni).
  - `socket_client`: il socket attraverso cui avviene la trasmissione dei comandi al server.

Funzionamento Dettagliato

- **Codifica e Invio del Comando**:
  - La funzione utilizza `comando.encode()` per convertire la stringa del comando in un formato di byte. Questo è necessario perché `sendto` richiede dati in forma di byte per l'invio.
  - Il comando codificato viene inviato al server utilizzando `socket_client.sendto`, specificando l'indirizzo del server (`HOST`, `PORT`).
- **Gestione degli Errori**:
  - È presente una gestione delle eccezioni `socket.error` che intercetta eventuali problemi di rete o connessione durante l'invio del comando. In caso di errore, viene stampato un messaggio descrittivo, con il dettaglio dell'errore (`e`).

Questa funzione è cruciale per permettere al client di interagire con il server, inviando comandi che saranno poi elaborati per aggiornare lo stato del gioco.

`muovi_navicella`

```
def muovi_navicella(event, socket_client, posizione_navicella, canvas, M, navicella_img, info_label):
    """
    Gestisce il movimento della navicella in risposta agli input
```

```
da tastiera.
    """
    global game_over
    if game_over:
        return  # Non fare nulla se il gioco è finito

    x, y = posizione_navicella
      # Determina la nuova posizione in base alla direzione del
tasto premuto
    if event.keysym == 'Up' and y > 0:
        nuova_posizione = (x, y - 1)
    elif event.keysym == 'Down' and y < M-1:
        nuova_posizione = (x, y + 1)
    elif event.keysym == 'Left' and x > 0:
        nuova_posizione = (x - 1, y)
    elif event.keysym == 'Right' and x < M-1:
        nuova_posizione = (x + 1, y)
    else:
        nuova_posizione = (x, y)

           disegna_navicella(canvas,   (x,   y),   nuova_posizione,
navicella_img)
             posizione_navicella[0],   posizione_navicella[1]   =
nuova_posizione

    # Controllo collisione tra navicella e meteorite
    if tuple(nuova_posizione) in meteoriti_posizioni:
        game_over = True
            canvas.create_text(150, 150, text="GAME OVER", font=
("Helvetica", 30), fill="red")
        return

    # Invia la nuova posizione della navicella al server
        invia_comando(f'{nuova_posizione[0]},{nuova_posizione[1]}',
socket_client)
```

Descrizione della Funzione

La funzione `muovi_navicella` gestisce il movimento della navicella in risposta ai comandi da tastiera, aggiornando la sua posizione nel gioco e inviando la nuova posizione al server.

- **Parametri**:
  - `event`: evento generato dalla pressione di un tasto, utilizzato per determinare la direzione del movimento.
  - `socket_client`: il socket utilizzato per inviare la nuova posizione al server.

- `posizione_navicella`: una lista contenente le coordinate attuali della navicella.
- `canvas`: il canvas di Tkinter dove sono visualizzati navicella e meteoriti.
- `M`: dimensione massima della griglia (numero di celle).
- `navicella_img`: immagine della navicella da visualizzare sul canvas.
- `info_label`: etichetta di Tkinter per visualizzare informazioni di gioco.

Funzionamento Dettagliato

- **Controllo dello Stato di Gioco**:
  - La funzione verifica se `game_over` è impostato a `True`. In tal caso, la funzione termina immediatamente senza eseguire ulteriori azioni.
- **Determinazione della Nuova Posizione**:
  - In base al tasto premuto (`Up`, `Down`, `Left`, `Right`), la funzione calcola la nuova posizione della navicella all'interno dei limiti della griglia. Se il tasto premuto non comporta un movimento valido, la navicella mantiene la posizione attuale.
- **Aggiornamento della Grafica**:
  - La funzione chiama `disegna_navicella` per aggiornare la posizione della navicella sul canvas. Successivamente, aggiorna `posizione_navicella` con le nuove coordinate.
- **Controllo delle Collisioni**:
  - La funzione verifica se la nuova posizione della navicella coincide con quella di un meteorite. In caso di collisione, imposta `game_over` a `True` e visualizza "GAME OVER" al centro del canvas.
- **Invio della Posizione al Server**:
  - Infine, la funzione invia la nuova posizione al server utilizzando `invia_comando`, convertendo la posizione in una stringa con il formato `x,y`.

Questa funzione è essenziale per gestire il movimento della navicella e mantenere sincronizzati il client e il server riguardo alla posizione della navicella e agli eventi di gioco.

`retry_gioco`

```
def retry_gioco(socket_client, canvas, posizione_navicella,
navicella_img, meteorite_img, info_label):
```

```
    """
    Riavvia il gioco resettando tutte le variabili e la griglia.
    """
    global score, game_over, meteoriti_posizioni, timer
    game_over = True
    meteoriti_posizioni.clear()
    canvas.delete("all")  # Cancella tutti gli elementi dal canvas
    disegna_griglia(canvas, M)  # Ridisegna la griglia
    posizione_navicella[0], posizione_navicella[1] = M//2, M//2  # Riposiziona la navicella al centro
                disegna_navicella(canvas,    posizione_navicella, posizione_navicella, navicella_img)
     invia_comando('retry', socket_client)  # Invia il comando di retry al server
    score = 0
    game_over = False
    aggiorna_info(info_label)
    if timer is not None:
        info_label.after_cancel(timer)
     aggiorna_punteggio(info_label)  # Ricomincia l'aggiornamento del punteggio
    threading.Thread(target=ricevi_meteoriti, args=(socket_client, posizione_navicella, canvas, meteorite_img, info_label)).start()
```

Descrizione della Funzione

La funzione `retry_gioco` permette di riavviare il gioco ripristinando lo stato iniziale della navicella, della griglia e delle variabili, consentendo una nuova partita.

- **Parametri**:
  - `socket_client`: il socket utilizzato per inviare il comando di riavvio al server.
  - `canvas`: il canvas di Tkinter dove vengono visualizzati navicella e meteoriti.
  - `posizione_navicella`: una lista che contiene le coordinate della navicella, usata per riposizionarla al centro.
  - `navicella_img`: immagine della navicella utilizzata per visualizzare l'oggetto sul canvas.
  - `meteorite_img`: immagine del meteorite, utilizzata quando vengono ridisegnati.
  - `info_label`: etichetta di Tkinter usata per visualizzare informazioni di gioco.

Funzionamento Dettagliato

- **Reset delle Variabili di Gioco**:

    - La funzione imposta `game_over` a `True` e svuota la lista `meteoriti_posizioni` per rimuovere le posizioni salvate dei meteoriti. Inoltre, `score` viene azzerato.

- **Reset della Griglia e della Posizione della Navicella**:
    - Cancella tutti gli elementi dal canvas con `canvas.delete("all")`, quindi ridisegna la griglia tramite `disegna_griglia`. La navicella viene riposizionata al centro della griglia (`M//2, M//2`), e viene disegnata di nuovo tramite `disegna_navicella`.

- **Invio del Comando di Riavvio al Server**:
    - La funzione invia al server il comando `retry` utilizzando `invia_comando`, segnalando l'inizio di una nuova partita.

- **Aggiornamento delle Informazioni e del Punteggio**:
    - Aggiorna le informazioni mostrate all'utente con `aggiorna_info(info_label)` e azzera e ripristina l'aggiornamento del punteggio tramite `aggiorna_punteggio(info_label)`.

- **Riavvio del Thread di Ricezione dei Meteoriti**:
    - Viene creato un nuovo thread per `ricevi_meteoriti` per permettere al client di ricevere informazioni dal server sui nuovi meteoriti e aggiornare il canvas in tempo reale.

Questa funzione consente al client di ripristinare lo stato di gioco e avviare una nuova partita senza dover riavviare l'applicazione.

`invia_meteoriti`

```
def invia_meteoriti(socket_server):
    """
      Funzione che invia le posizioni dei meteoriti ai client connessi.
    """
    global meteoriti_posizioni
    while True:
           time.sleep(2)   # Attende 2 secondi tra ogni invio di meteoriti
        with lock:
            # Aggiunge nuovi meteoriti se non è stato raggiunto il numero massimo
            if len(meteoriti_posizioni) < n:
```

```
                    for _ in range(2):  # Genera 2 meteoriti ogni 2
secondi
                      if len(meteoriti_posizioni) < n:
                            posizione_meteorite = (random.randint(0,
M-1), random.randint(0, M-1))

meteoriti_posizioni.add(posizione_meteorite)

            # Calcola nuove posizioni per i meteoriti esistenti
            nuove_posizioni = set()
            for posizione in meteoriti_posizioni:
                nuova_posizione = (
                                    max(0,  min(M-1,  posizione[0]  +
random.choice([-1, 0, 1]))),
                                    max(0,  min(M-1,  posizione[1]  +
random.choice([-1, 0, 1])))
                )
                nuove_posizioni.add(nuova_posizione)
            meteoriti_posizioni = nuove_posizioni

               # Invia le nuove posizioni dei meteoriti a tutti i
client connessi
            for addr in client_addrs:
                    socket_server.sendto(b'clear', addr)  # Invia
comando per pulire i meteoriti
                for posizione in meteoriti_posizioni:
                              socket_server.sendto(f'{posizione[0]},
{posizione[1]}'.encode(), addr)
                            time.sleep(0.05)  # Aggiunge un piccolo
ritardo per evitare pacchetti UDP troppo frequenti
```

Descrizione della Funzione

La funzione `invia_meteoriti` invia le posizioni aggiornate dei meteoriti a tutti i client connessi. È progettata per aggiungere nuovi meteoriti periodicamente e per aggiornare le posizioni di quelli esistenti.

- **Parametri**:
  - `socket_server`: il socket server che invia le posizioni dei meteoriti ai client.

Funzionamento Dettagliato

- **Ciclo di Invio Continuo**:
  - La funzione è progettata per funzionare in un ciclo continuo, inviando le posizioni aggiornate ogni 2 secondi per mantenere il gioco sincronizzato.

- **Aggiunta di Nuovi Meteoriti**:
    - Se il numero di meteoriti attuali è inferiore al massimo consentito (`n`), vengono aggiunti fino a 2 nuovi meteoriti. Ogni nuova posizione viene generata casualmente all'interno dei limiti della griglia.
- **Aggiornamento delle Posizioni dei Meteoriti**:
    - Le posizioni dei meteoriti vengono aggiornate in modo casuale per ciascuna iterazione, con uno spostamento di uno o zero passi in direzione casuale. Le nuove posizioni vengono raccolte in `nuove_posizioni` e sostituiscono quelle precedenti.
- **Invio delle Posizioni ai Client**:
    - Per ogni client connesso, la funzione invia un comando `clear` per cancellare i meteoriti precedenti, seguito da un invio sequenziale delle nuove posizioni. Un piccolo ritardo (`time.sleep(0.05)`) è aggiunto tra ogni invio per evitare sovraccarichi di pacchetti.

Questa funzione permette di aggiornare continuamente la posizione dei meteoriti in tempo reale, mantenendo sincronizzati il server e i client.

`ricevi_client`

```
def ricevi_client(socket_server):
    """
    Funzione che gestisce la ricezione dei comandi dai client.
    """
    global client_addrs
    while True:
        try:
            # Riceve dati dal client
            dati, addr = socket_server.recvfrom(1024)
            comando = dati.decode()

            if comando == 'start':
                      # Aggiunge il client all'elenco dei client
connessi
                with lock:
                    client_addrs.add(addr)
                     socket_server.sendto(b'Benvenuto nel gioco!',
addr)
            elif comando == 'stop':
                      # Rimuove il client dall'elenco dei client
connessi
```

```
                with lock:
                    client_addrs.remove(addr)
            elif comando == 'retry':
                 # Pulisce le posizioni dei meteoriti per il retry
del gioco
                with lock:
                    meteoriti_posizioni.clear()
        except socket.error as e:
            print(f"Errore durante la ricezione dei dati: {e}")
        except Exception as e:
            print(f"Errore sconosciuto: {e}")
```

Descrizione della Funzione

La funzione `ricevi_client` gestisce la ricezione dei comandi dai client connessi, aggiungendoli o rimuovendoli dalla lista dei client attivi e gestendo il comando di riavvio.

- **Parametri**:
  - `socket_server`: il socket server usato per ricevere i comandi dai client.

Funzionamento Dettagliato

- **Ricezione dei Dati dal Client**:
  - La funzione è progettata per eseguire un ciclo continuo, ricevendo pacchetti di dati da ciascun client. I dati ricevuti sono decodificati in stringa e memorizzati nella variabile `comando`.
- **Gestione dei Comandi del Client**:
  - **Comando `'start'`**: Aggiunge l'indirizzo del client (`addr`) alla lista `client_addrs` e invia un messaggio di benvenuto al client.
  - **Comando `'stop'`**: Rimuove l'indirizzo del client da `client_addrs`, segnalando la disconnessione del client.
  - **Comando `'retry'`**: Cancella le posizioni dei meteoriti (`meteoriti_posizioni.clear()`), consentendo un riavvio del gioco.
- **Gestione degli Errori**:
  - La funzione gestisce eventuali errori di rete (`socket.error`) e altri errori generici, con messaggi di errore specifici per il debug.

Questa funzione consente al server di monitorare e gestire i client connessi, eseguendo azioni specifiche in base ai comandi ricevuti per garantire la sincronizzazione del gioco.

3. Casi d'Uso

Caso d’Uso 1: Connessione al Server

- **Attori coinvolti**:
  - Client (Navicella)
  - Server (Gestore dell'ambiente con meteoriti)
- **Obiettivo**:
  - Connettere il client al server per iniziare una nuova sessione di gioco.
- **Sequenza di azioni**:
  1. Il client invia un comando `start` al server tramite la funzione `invia_comando`.
  2. Il server riceve il comando tramite la funzione `ricevi_client`.
  3. Il server aggiunge l’indirizzo del client alla lista `client_addrs` e invia un messaggio di benvenuto.
  4. Il client riceve il messaggio e visualizza l’inizio della sessione di gioco.
- **Condizioni iniziali**:
  - Il server è attivo e in ascolto dei comandi in ingresso.
  - Il client è configurato per comunicare con il server.
- **Condizioni finali**:
  - Il client è connesso al server e pronto per ricevere aggiornamenti di gioco.

Caso d’Uso 2: Movimento della Navicella

- **Attori coinvolti**:
  - Client (Navicella)
- **Obiettivo**:
  - Consentire al giocatore di spostare la navicella per evitare i meteoriti e navigare nell’ambiente.
- **Sequenza di azioni**:
  1. Il client rileva un comando di movimento da tastiera (ad esempio, una freccia direzionale).

2. La funzione `muovi_navicella` calcola la nuova posizione in base al tasto premuto.
3. Se la posizione è valida, `muovi_navicella` aggiorna la posizione sul canvas e invia la nuova posizione al server tramite `invia_comando`.
4. Il server aggiorna la posizione della navicella per il gioco e la sincronizzazione con gli altri componenti.

- **Condizioni iniziali**:
  - La sessione di gioco è attiva.
  - La navicella è posizionata nella griglia.
- **Condizioni finali**:
  - La navicella si è spostata nella nuova posizione oppure è rimasta ferma (se si trova ai limiti della griglia).

Caso d'Uso 3: Invio delle Posizioni dei Meteoriti

- **Attori coinvolti**:
  - Server
- **Obiettivo**:
  - Aggiornare continuamente i client con le nuove posizioni dei meteoriti per mantenere il gioco sincronizzato.
- **Sequenza di azioni**:
  1. Il server esegue la funzione `invia_meteoriti`, che genera nuove posizioni per i meteoriti e aggiorna quelle esistenti.
  2. Ogni posizione viene inviata ai client connessi tramite `socket_server.sendto`, con un comando di "clear" per aggiornare le posizioni esistenti.
  3. I client ricevono le nuove posizioni e aggiornano la visualizzazione dei meteoriti sul canvas.
- **Condizioni iniziali**:
  - La sessione di gioco è attiva e i client sono connessi.
  - Il server ha almeno una posizione di meteorite da inviare.
- **Condizioni finali**:
  - I client hanno aggiornato le posizioni dei meteoriti sul canvas.

Caso d’Uso 4: Gestione del Game Over

- **Attori coinvolti**:
  - Client
  - Server
- **Obiettivo**:
  - Rilevare una collisione e terminare la sessione di gioco per evitare movimenti o aggiornamenti ulteriori.
- **Sequenza di azioni**:
  1. Il client rileva una collisione tra la navicella e un meteorite tramite la funzione `muovi_navicella`.
  2. La funzione imposta `game_over` a `True` e visualizza un messaggio "GAME OVER" sul canvas, interrompendo ogni ulteriore movimento della navicella.
- **Condizioni iniziali**:
  - Il client è connesso e la navicella si trova sulla griglia.
  - I meteoriti sono presenti sulla griglia e in movimento.
- **Condizioni finali**:
  - La partita è terminata con il messaggio "GAME OVER" visualizzato sul canvas, e la navicella non può più muoversi.

4. Gestione degli Errori e Tolleranza ai Guasti

Nel progetto di rete client-server, la gestione degli errori è fondamentale per garantire la stabilità e la continuità del gioco, soprattutto considerando l'uso del protocollo UDP, che non garantisce l'affidabilità dei pacchetti. La struttura del codice include diversi meccanismi per la gestione degli errori e per minimizzare i possibili guasti di rete.

1. Gestione degli Errori di Rete

- **Eccezioni Socket**:
  - Sia il client che il server utilizzano il modulo `socket` per gestire la comunicazione di rete. Poiché UDP non prevede una conferma di ricezione, eventuali problemi di connessione o di trasmissione possono verificarsi in qualsiasi momento.

- Nella funzione `ricevi_client`, ad esempio, se si verifica un errore durante la ricezione dei dati (`socket.error`), viene catturato e gestito un messaggio di errore specifico. Questo messaggio viene visualizzato sulla console, consentendo di identificare l'indirizzo o il tipo di errore occorso. In modo simile, la funzione `invia_comando` gestisce gli errori di trasmissione con un blocco `try-except`, mantenendo il client in esecuzione anche in caso di errore temporaneo.

2. Tolleranza ai Guasti e Riconnessione

- **Gestione dei Client Disconnessi**:
  - Quando un client si disconnette volontariamente o perde la connessione, il server aggiorna automaticamente la lista `client_addrs`, rimuovendo il client attraverso il comando `stop`. Questa rimozione viene eseguita nel blocco `try-except` della funzione `ricevi_client`, garantendo che il server continui a funzionare anche quando un client lascia improvvisamente il gioco.
- **Ripristino del Gioco**:
  - Nel caso di riavvio del gioco da parte di un client (con il comando `retry`), il server riceve una richiesta di "pulizia" tramite `meteoriti_posizioni.clear()`. Questo assicura che le posizioni dei meteoriti siano ripristinate, consentendo una nuova sessione senza riavviare il server.

3. Strategie per Mitigare i Rischi di Perdita di Pacchetti

- **Invio Periodico dei Meteoriti con Ritardo**:
  - Per evitare l'eccessivo carico di pacchetti UDP sulla rete, la funzione `invia_meteoriti` utilizza un ritardo di 0,05 secondi tra l'invio delle coordinate di ciascun meteorite ai client. Questa ottimizzazione riduce il rischio di sovraccarico e permette una trasmissione dei pacchetti più regolare, limitando la probabilità di pacchetti persi.
  - Il server invia inoltre un comando `clear` prima di ogni nuovo set di posizioni, così che i client aggiornino la posizione dei meteoriti in modo sincrono e possano ignorare i pacchetti eventualmente persi o fuori sequenza.

4. Log e Feedback in Console

- **Messaggi di Log**:
  - Gli errori di connessione e altre eccezioni sono registrati con messaggi di log, che permettono di identificare eventuali problemi di connessione e di monitorare l'affidabilità della rete. Ad esempio, i messaggi di errore generati

da `socket.error` aiutano a diagnosticare problemi specifici senza interrompere l'esecuzione del server o del client.

La gestione degli errori e le misure di tolleranza ai guasti implementate garantiscono che il sistema possa gestire disconnessioni inattese e problemi temporanei di rete senza interrompere la sessione di gioco. Grazie a queste misure, il sistema risponde con continuità, consentendo ai giocatori di riconnettersi e riprendere la sessione senza dover riavviare il server.

5. Analisi delle Performance e Ottimizzazione

Le tecniche di ottimizzazione adottate in questo progetto puntano a garantire velocità e reattività, fondamentali per un gioco in tempo reale. Di seguito vengono analizzati gli aspetti principali relativi alle prestazioni di rete e alle ottimizzazioni implementate nel sistema.

1. Controllo della Frequenza di Invio dei Pacchetti

- **Ritardo per Prevenire Sovraccarico**:
  - All'interno della funzione `invia_meteoriti`, viene applicato un breve ritardo (`time.sleep(0.05)`) tra l'invio dei pacchetti delle coordinate dei meteoriti. Questo piccolo intervallo limita la trasmissione dei dati, riducendo il rischio di congestionare la rete.

2. Sincronizzazione con Lock per Dati Condivisi

- **Consistenza nei Dati Condivisi**:
  - Il server utilizza un `lock` per proteggere l'accesso ai dati condivisi tra thread, come `client_addrs` e `meteoriti_posizioni`. Questo assicura che le modifiche ai dati non si sovrappongano, evitando problemi di race condition e garantendo un accesso sicuro e consistente.

3. Threading per Migliorare l'Efficienza

- **Gestione Multithreaded delle Operazioni del Server**:
  - Per ottimizzare le prestazioni, il server gestisce la ricezione dei comandi e l'invio delle posizioni dei meteoriti su thread separati. Ciò permette di rispondere rapidamente alle richieste dei client senza interrompere il flusso di aggiornamenti di gioco.

Queste ottimizzazioni migliorano l’esperienza di gioco rendendola fluida e reattiva, bilanciando le risorse di rete in modo efficiente.