Università degli Studi di Napoli "Parthenope"
Corso di laurea in Informatica (percorso Generale)

Corso di Reti di Calcolatori e Laboratorio

Network Pong

Docente
Emanuel Di Nardo

Candidato
Raffaele Talente
0124002658

Anno Accademico 2023–2024

Contents

1 Descrizione del progetto

L'obiettivo del progetto è quello di implementare un gioco basato su **Pong**, tramite comunicazione in tempo reale con UDP.
Pong, uno dei primissimi videogiochi arcade, è un simulatore di ping-pong bidimensionale: i due giocatori controllano una racchetta ciascuno, situate ai due lati dello schermo, e si scambiano una pallina che si muove da un lato all'altro dello schermo. Nel momento in cui un giocatore non riesce a intercettare la pallina l'avversario guadagna un punto.

Il progetto è stato sviluppato utilizzando il linguaggio di programmazione **Python**. La griglia di gioco è stata rappresentata come una griglia di caratteri, che possono essere vuoti, occupati da una racchetta oppure occupati dalla pallina. E' stata inoltre utilizzata la libreria **Pygame** per gestire l'input da tastiera dell'utente e la rappresentazione a schermo della griglia di gioco.

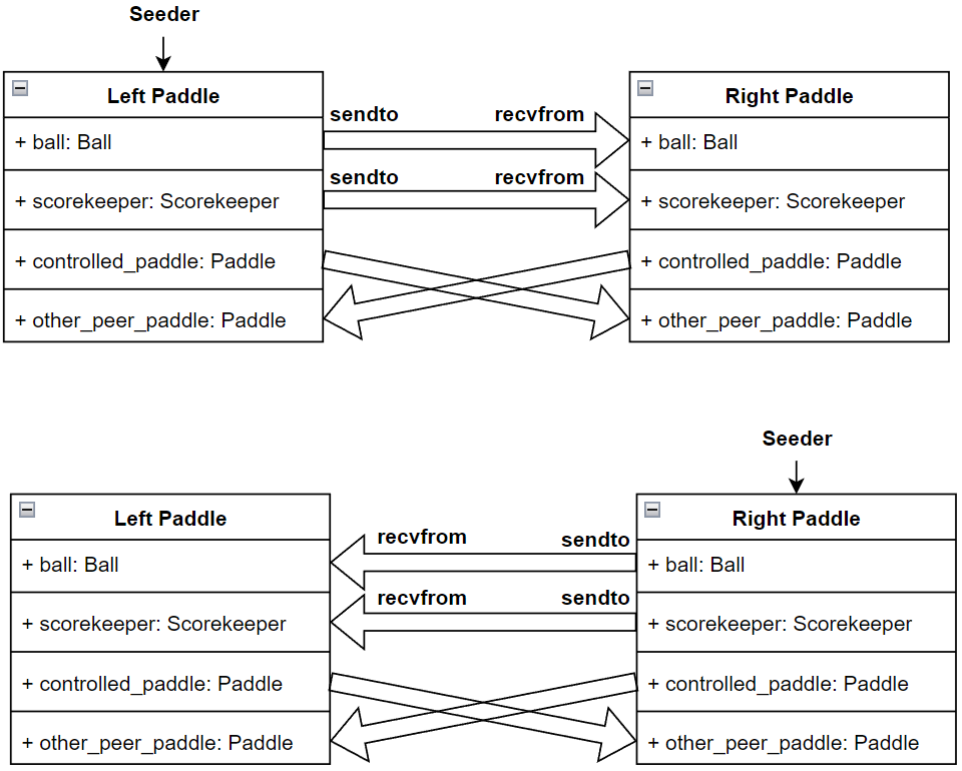

Figure 1: Schema dell'architettura

2 Schema dell'architettura

L'architettura del progetto è basata su una rete P2P puramente decentralizzata, in quanto tutti i nodi sono uguali, sebbene quantitativamente piccola:
I due giocatori costituiscono i **Peer**, poiché svolgono la funzione sia di client che di server: entrambi devono notificare immediatamente la controparte riguardo i movimenti della propria racchetta, inoltre l'ultimo giocatore che ha toccato la pallina deve fare da *seeder*, deve cioè calcolare il movimento della pallina e comunicarlo alla controparte. Il seeder si occupa anche di aggiornare il punteggio e di comunicarlo all'altro giocatore.

3 Dettagli implementativi

3.1 Implementazione dei Peer

La classe Peer è stata progettata nel seguente modo:

- Il costruttore della classe crea la socket UDP ed effettua il bind su tutte le interfacce, utilizzando una porta di default arbitaria. Vengono poi create le variabili degli oggetti utilizzati all'interno del gioco, l'implementazione delle classi di tali oggetti è presente nel file *app/components.py*
- L'indirizzo dell'altro peer viene passato come argomento da linea di comando ed inserito opportunamente.
- Il metodo send_data trasforma l'oggetto Python da inviare in uno stream di byte attraverso il modulo *pickle*.
- Il metodo *receive_data* gestisce l'ottenimento dei dati dalla *recvfrom*, trasformando lo stream di byte in un oggetto Python.
  E' importante inserire un timeout, in quanto è imperativo che la recvfrom non blocchi indefinitivamente il processo: il timeout è stato impostato tramite una costante a *5.5 ms*, questo numero deriva dal fatto che se si vuole mantenere il gioco a 60 *frame al secondo*, è necessario che ogni frame non richieda più di *16.6 ms*. Poiché in ogni frame un Peer al più deve ricevere informazioni sulla pallina, sulla racchetta dell'altro giocatore e sullo score, il valore massimo di attesa non può essere più elevato.
- Il metodo *receive_and_replace_object_data* richiama il metodo *receive_data* ed effettua l'azione appropriata a seconda dell'oggetto ritornato.
  - Se viene ritornato un oggetto, si aggiorna opportunamente la variabile relativa.
  - Se la chiamata va in *timeout*, la funzione ritorna.
    Il timeout viene effettivamente ignorato in quanto è importante che il gioco prosegui, uno spostamento di un frame mancato non è particolarmente impattante sull'applicazione rispetto ad un processo che si blocca per assicurarsi che ogni pacchetto inviato sia ricevuto.

```
# import omessi per semplicità

    class Peer:
def __init__(self, paddle_id):
    # viene creata una socket associata al peer
    # il bind viene effettuato su '', che è l'equivalente di INADDR_ANY
    self.peer_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    # come porta viene utilizzata una porta di default arbitaria
    # a cui viene aggiunto l'id del paddle
    self.peer_socket.bind(('', DEFAULT_PORT + paddle_id))

    # l'id è 0 (racchetta sinistra) o 1 (racchetta destra)
    self.id = paddle_id

    # inizializzazione delle variabili utilizzate dalla classe
    self.other_peer = ('', DEFAULT_PORT + paddle_id)
    self.controlled_paddle = Paddle()
    self.other_peer_paddle = Paddle()
    self.ball = Ball()
    self.scorekeeper = Scorekeeper()

# l'indirizzo dell'altro seed viene passato
# come argomento da riga di comando
def set_other_peer(self, address):
    if self.id == LEFT_PADDLE_ID:
        self.other_peer = (address, DEFAULT_PORT + RIGHT_PADDLE_ID)
    else:
        self.other_peer = (address, DEFAULT_PORT + LEFT_PADDLE_ID)
```

Figure 2: Implementazione del costruttore della classe Peer

– Se vi è un altro errore, solitamente causato dalla mancanza dell'altro peer, il processo semplicemente va in sleep per un secondo prima di ritentare.

```
# invia dati all'altro peer
# trasforma l'oggetto in uno stream di byte con pickle
def send_data(self, object_data):
    data = pickle.dumps(object_data)
    self.peer_socket.sendto(data, self.other_peer)

# riceve dati dall'altro peer dopo aver inserito il timeout
# ritorna un oggetto Python ottenuto dallo stream di byte con pickle
def receive_data(self):
    self.peer_socket.settimeout(TIMEOUT)
    data, address = self.peer_socket.recvfrom(MAX_BUF)
    object_data = pickle.loads(data)
    return object_data

# ottiene un oggetto chiamando il metodo receive_data
# ed effettua l'operazione appropriata
def receive_and_replace_object_data(self):
    object_data = object
    try:
        object_data = self.receive_data()

    # se la recvfrom va in timeout la funzione semplicemente ritorna
    except socket.timeout:
        return

    # se l'altro processo non è presente, aspetta
    except socket.error:
        print('waiting for the other peer...')
        time.sleep(1)

    # se riceve un oggetto correttamente
    # effettua l'operazione appropriata
    if type(object_data) is Ball:
        self.ball = object_data
    elif type(object_data) is Paddle:
        self.other_peer_paddle = object_data
    elif type(object_data) is Scorekeeper:
        self.scorekeeper = object_data
```

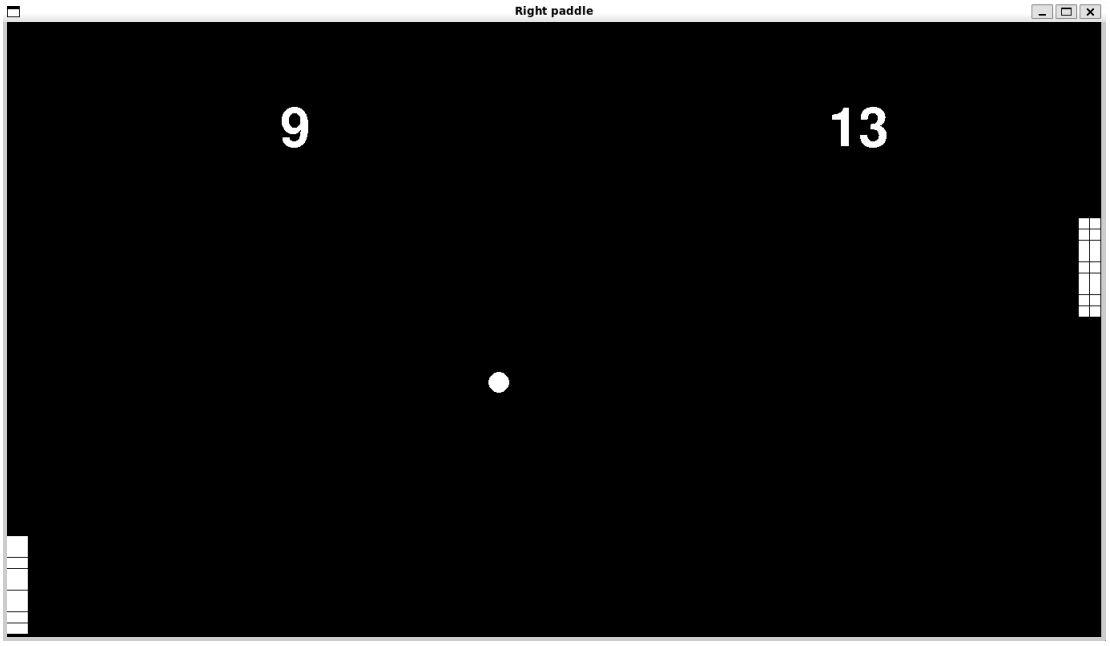

Figure 3: Immagine dell'interfaccia di gioco

4 Il gioco

Il gioco viene avviato tramite linea di comando passando opzionalmente l'indirizzo dell'altro peer (se questi non viene inserito, l'applicazione utilizza l'indirizzo di loopback).
Dopo aver passato gli argomenti ottenuti da linea di comando, il programma richiama la funzione *peer_run* che è divisa in due parti:

- **Creazione del peer e delle variabili di gioco** (Figura 4) Vengono inizializzate tutte le variabili necessarie al funzionamento della partita, tra cui la griglia di gioco, la classe del peer locale e le variabili necessarie al funzionamento di *pygame*
- **Il game loop** (Figura 5), ovvero il loop del gioco vero e proprio, composto come segue:
  - Ottenimento dell'input da parte dell'utente, il tasto W muove la racchetta verso l'alto, mentre il tasto S la muove verso il basso.
  - Invio del pacchetto UDP contenente le informazioni sulla propria racchetta e ricezione del pacchetto della controparte.
  - A questo punto, il seeder gestisce opportunamente il movimento della pallina, se la pallina esce da un lato della griglia,

```
# Import omessi per semplicità

def peer_run(args):
    # inizializzazione delle variabili
    grid = Grid()
    pygame.init()
    pygame.font.init()
    screen = pygame.display.set_mode((SCREEN_WIDTH, SCREEN_HEIGHT))
    clock = pygame.time.Clock()

    # creazione del peeer
    peer = Peer(args.peer_id)
    peer.set_other_peer(args.other_peer_address)

    # LEFT_PADDLE_ID è 0
    if peer.id == LEFT_PADDLE_ID:
        pygame.display.set_caption("Left paddle")
    else:
        pygame.display.set_caption("Right paddle")

    # continua...
```

Figure 4: Inizializzazione del gioco

procede ad aggiornare opportunamente lo scorekeeper, e invia i dati alla controparte.
Nel mentre l'altro peer si limita ad ottenere i dati dalla controparte.

– Infine, entrambi i peer aggiornano la propria griglia in base ai dati posseduti e aggiornano il display.

```
# main game loop
# alcune parti omesse per semplicità
while True:
    # gestione dell'input
        # se viene premuto W...
        peer.controlled_paddle.velocity = -1
        # se viene premuto S...
        peer.controlled_paddle.velocity = 1

    # scambio dei dati delle proprie racchette
    peer.controlled_paddle.move()
    peer.send_data(peer.controlled_paddle)
    peer.receive_and_replace_object_data()

    if peer.ball.seeder_id == peer.id:
        # la pallina viene mossa oportunamente
        # e la posizione aggiornata viene inviata
        peer.send_data(peer.ball)
        if peer.ball.is_out():
            if peer.ball.pos_x <= 0:
                peer.scorekeeper.add_score(left_paddle_scored=False)
            elif peer.ball.pos_x >= GRID_WIDTH:
                peer.scorekeeper.add_score(left_paddle_scored=True)
            peer.ball.reset_position()
            peer.ball.set_random_velocity()
            peer.send_data(peer.ball)
            peer.send_data(peer.scorekeeper)
    # l'altro peer ottiene semplicemente i dati
    else:
        peer.receive_and_replace_object_data()
        peer.receive_and_replace_object_data()

    # aggiornamento della griglia,
    # del codice è stato omesso per semplicità
    grid.update_grid(  # args  )
    grid.display(screen)
    peer.scorekeeper.display(screen)
    pygame.display.flip()
    clock.tick(FPS)
```

Figure 5: Il game loop

5 Manuale istruzioni sull'esecuzione

Assicurarsi di avere Python installato su ogni macchina, il progetto è stato sviluppato con Python 3.11.4

L'applicazione può essere eseguita come segue:

- Clonare la repository:

```
git clone
https://github.com/rt2013G/reti9-network-pong.git
&& cd reti9-network-pong
```

- Creare un virtual environment ed attivarlo:

```
virtualenv -p python3 ./venv
&& source ./venv/bin/activate
```

- Installare i package necessari:

```
pip install -r requirements.txt
```

- Avviare l'applicazione sulla macchina su cui utilizzare la racchetta di **destra**.
  Questo passaggio è importante in quanto di default il seeder iniziale è la racchetta sinistra.

```
python3 pong.py 1 <IP controparte>
```

  oppure, se i due processi si trovano sulla stessa macchina:

```
python3 pong.py 1
```

  L'applicazione inserirà automaticamente l'indirizzo di loopback in questo caso.

- Avviare l'applicazione sulla macchina su cui utilizzare la racchetta di **sinistra**.
  Attenzione: il gioco inizierà automaticamente

  ```
  python3 pong.py 0 <IP controparte>
  ```

  oppure, se i due processi si trovano sulla stessa macchina:

  ```
  python3 pong.py 0
  ```

  L'applicazione inserirà automaticamente l'indirizzo di loopback in questo caso.