BỘ GIÁO DỤC VÀ ĐÀO TẠO

ĐẠI HỌC KINH TẾ TP HỒ CHÍ MINH

UEH UNIVERSITY

BÁO CÁO ĐỒ ÁN CUỐI KỲ

XÂY DỰNG GAME CARO BẰNG NGÔN NGỮ HASKELL

Môn: Lập trình hàm

Giáo viên: Hồ Thị Thanh Tuyến

Sinh viên thực hiện - MSSV:

1. Võ Lê Khánh Linh - 31231021529
2. Nguyễn Thảo Trân - 31231026116

Lớp - Khóa: CS0001 - K49

Mã lớp học phần: 25C1INF50915401

TP. Hồ Chí Minh, ngày 25 tháng 10 năm 2025

LỜI CẢM ƠN

Kính gửi cô Hồ Thị Thanh Tuyến,

Chúng em xin được gửi lời cảm ơn sâu sắc đến cô – người đã tận tình hướng dẫn và giảng dạy trong suốt quá trình học tập cũng như khi thực hiện tiểu luận cuối kỳ này.

Qua mỗi buổi học, cô không chỉ truyền đạt kiến thức mà còn giúp chúng em dần hình thành được cách tiếp cận vấn đề một cách khoa học và logic. Những kiến thức và góp ý của cô là định hướng quan trọng để chúng em hoàn thành đề tài một cách nghiêm túc và có hệ thống hơn.

Trong quá trình thực hiện tiểu luận cuối kỳ, chúng em đã có cơ hội vận dụng những kiến thức nền tảng mà cô truyền đạt để áp dụng vào một đề tài cụ thể. Qua đó, chúng em nhận ra rằng việc học không dừng lại ở lý thuyết mà còn là sự liên hệ với thực tiễn và khả năng tự giải quyết vấn đề.

Một lần nữa, chúng em xin được cảm ơn cô vì sự tận tình, nghiêm túc và tâm huyết trong giảng dạy. Đây sẽ là những trải nghiệm quý giá, là hành trang đáng nhớ trong chặng đường học tập của chúng em.

Trân trọng cảm ơn cô,

Võ Lê Khánh Linh, Nguyễn Thảo Trân

MỤC LỤC

I. GIỚI THIỆU ĐỒ ÁN

1. Tổng quan về đề tài, mục tiêu và phạm vi nghiên cứu

Lập trình hàm là một trong những mô hình tư duy quan trọng trong khoa học máy tính. Bên cạnh lập trình hướng đối tượng, lập trình hàm tập trung vào tính thuần khiết (purity), bất biến (immutability) và các cấu trúc trừu tượng bậc cao thay vì thay đổi trạng thái hay dùng vòng lặp như các mô hình khác. Tuy nhiên, việc áp dụng các khái niệm này vào các ứng dụng trong thế giới thực vốn dĩ luôn tồn tại trạng thái và các tác vụ I/O (Input/Output) không thể đoán trước được là một thách thức lớn.

Đồ án sẽ xây dựng game caro 3x3 theo mô hình client-server, cho phép hai người chơi kết nối và thi đấu với nhau. Mục tiêu chính không chỉ là tạo ra một sản phẩm game, mà còn là áp dụng trực tiếp các kiến thức lập trình hàm đã học để thiết kế và cài đặt trò chơi bằng ngôn ngữ Haskell, đồng thời khảo sát và đánh giá tính ứng dụng của Haskell trong lĩnh vực lập trình hàm và lập trình song song thông qua các điểm kỹ thuật sau:

- Xử lý I/O: áp dụng cấu trúc Monad IO để bao bọc và kiểm soát các tác vụ I/O không thuần khiết (network I/O, console I/O), đảm bảo rằng các hàm xử lý luật chơi cốt lõi vẫn giữ được tính thuần túy.
- Quản lý trạng thái: quản lý trạng thái chung của ván đấu (bàn cờ, lượt đi) giữa nhiều luồng của server.
- Xử lý đồng bộ hóa: giải quyết các vấn đề kinh điển về concurrency (như lỗi tranh chấp dữ liệu) một cách an toàn và hiệu quả.
- Thực hiện cấu trúc client-server: xây dựng một kiến trúc mạng phân tán sử dụng giao thức TCP/IP (socket), trong đó server đóng vai trò điều phối và quản lý logic game, còn client đảm nhận vai trò giao diện người dùng và tương tác.

Phạm vi của đồ án được xác định rõ ràng từ ban đầu là tập trung tối đa vào các thách thức kỹ thuật của lập trình song song và quản lý trạng thái, chứ không mở rộng quy mô sản phẩm. Cụ thể, phiên bản cờ caro người chơi sẽ chơi là trên bàn cờ 3x3, trong đó người chơi sẽ giành chiến thắng khi có được ba quân cờ liên tiếp theo chiều ngang, dọc hoặc chéo.

Về mặt chức năng, hệ thống phải hỗ trợ đầy đủ logic cốt lõi của ván đấu, bao gồm xác định chính xác trạng thái thắng/thua, trạng thái hòa, đồng thời xử lý các tình huống bất thường bằng cách báo lỗi khi người chơi đi sai lượt hoặc nhập sai cú pháp lệnh. Ngoài ra, đồ án còn

bao gồm cơ chế chơi lại (rematch) sau khi kết thúc một ván, yêu cầu logic đồng bộ hóa phức tạp hơn.

Về môi trường giao tiếp, giao diện người dùng được xây dựng hoàn toàn bằng dòng lệnh (console/terminal). Thiết kế này giúp tối giản hóa tầng giao diện để tập trung vào việc phát triển giao thức giao tiếp tối giản thông qua socket. Quan trọng nhất, để tập trung phân tích và đánh giá cơ chế STM, server được giới hạn chỉ quản lý và điều phối một cặp đấu duy nhất tại một thời điểm, loại bỏ yêu cầu phát triển sảnh chờ cho nhiều trận đấu song song.

2. Các thư viện sử dụng

Công nghệ/Thư viện	Mục đích sử dụng	Vai trò
Ngôn ngữ Haskell	Ngôn ngữ lập trình chính	Bắt buộc tách biệt logic game thuần túy khỏi logic hệ thống (mạng, trạng thái thay đổi). Việc này giúp giảm thiểu lỗi và tối ưu hóa quá trình kiểm thử.
Monad IO	Xử lý các tác vụ I/O	Monad IO được sử dụng để tuần tự hóa các hành động như nhận tin nhắn từ client, gửi phản hồi, và in trạng thái ra console.
Network.Socket	Lập trình mạng TCP/IP	Cung cấp các hàm I/O cấp thấp để thiết lập kết nối socket trên server (lắng nghe trên một cổng) và trên client (kết nối tới địa chỉ server).

stm (software transactional memory)	Quản lý đồng bộ hóa	Công nghệ trung tâm cho server. STM được sử dụng thông qua kiểu dữ liệu TVar để lưu trữ trạng thái game. TVar cho phép các luồng truy cập và cập nhật trạng thái game trong một giao dịch an toàn, tự động loại bỏ lỗi tranh chấp dữ liệu mà không cần cơ chế khóa phức tạp.
Control.Concurrent.sync	Lập trình song song	Được áp dụng trên client để chạy song song hai tác vụ: 1) Nhận dữ liệu liên tục từ server và 2) Chờ người dùng nhập lệnh. Hàm race_ được sử dụng để quản lý hai luồng này một cách hiệu quả.
Giao thức giao tiếp tùy chỉnh	Trao đổi dữ liệu	Xây dựng một bộ quy tắc giao tiếp tối giản (dạng chuỗi) để client và server hiểu nhau (ví dụ: MOVE:1,2). Điều này đảm bảo tính đồng nhất của dữ liệu được truyền qua socket.

II. PHÂN TÍCH VÀ THIẾT KẾ HỆ THỐNG

1. Mô hình Client-Server

Mô hình client–server tập trung được lựa chọn làm mô hình nền tảng cho đồ án, trong đó server đóng vai trò trung tâm chịu trách nhiệm điều phối, xử lý logic và duy trì trạng thái hệ thống, còn client đảm nhận nhiệm vụ tương tác trực tiếp với người dùng. Cách tổ chức này cho phép tách biệt hoàn toàn các vấn đề liên quan đến đồng bộ hóa và quản lý trạng thái, từ đó giúp hệ thống đạt được tính ổn định, dễ mở rộng và phù hợp với mục tiêu nghiên cứu về điều phối đa luồng trong môi trường song song.

a. Vai trò và cơ chế đa luồng của server

Server được thiết kế để hoạt động như một trọng tài trò chơi, chịu trách nhiệm giám sát và điều phối toàn bộ tiến trình của ván đấu. Việc triển khai được thực hiện dựa trên thư viện *Network.Socket*, server thiết lập một điểm lắng nghe cố định tại cổng *3000*, cho phép tiếp nhận kết nối từ các client thông qua giao thức TCP. Khi nhận đủ hai kết nối từ người chơi, server sẽ khởi tạo một phiên đấu (*gameSession*), trong đó toàn bộ logic điều khiển được vận hành trong một không gian luồng độc lập.

Quá trình khởi tạo phiên đấu được thực hiện bằng cơ chế sinh luồng của Haskell thông qua hàm *forkFinally*. Sau khi chấp nhận hai kết nối TCP, server sử dụng lời gọi *forkFinally* (*gameSession connections*) để tạo một luồng mới phụ trách toàn bộ quá trình điều phối giữa hai người chơi. *forkFinally* không chỉ giúp phân tách quá trình xử lý độc lập mà còn đảm bảo quản lý tài nguyên an toàn, hàm dọn dẹp tài nguyên *close conn1 >> close conn2* luôn được thực thi khi luồng kết thúc, dù phiên đấu kết thúc bình thường hay do ngoại lệ bất thường.

Bên trong *gameSession*, server triển khai cơ chế điều phối luồng song song bằng cách khởi chạy hai tiến trình I/O đồng thời sử dụng *race_*, mỗi tiến trình tương ứng với một người chơi X và O thông qua hàm *playerListener*. Hai tiến trình này hoạt động song song, cùng truy cập và thao tác trên cùng một trạng thái trò chơi (*GameState*) được bọc bởi *TVar*. Cơ chế STM của Haskell đảm bảo rằng mọi thao tác ghi hoặc cập nhật trạng thái đều được thực hiện theo nguyên tắc nguyên tử, loại bỏ hoàn toàn khả năng xảy ra lỗi tranh chấp dữ liệu. Khi một trong hai người chơi ngắt kết nối hoặc thoát khỏi trò chơi, cơ chế đồng bộ của *race_* giúp đảm bảo rằng luồng còn lại được hủy bỏ ngay lập tức, từ đó kết thúc phiên đấu một cách gọn gàng mà không để lại tiến trình rác. Nhờ đó, server duy trì được tính ổn định và khả năng phản hồi cao

trong môi trường đa người chơi, đồng thời giữ vững nguyên tắc bất biến và toàn vẹn dữ liệu của hệ thống.

b. Vai trò và cơ chế đồng thời của client

Client được thiết kế với mục tiêu tối ưu hóa trải nghiệm người dùng trên giao diện dòng lệnh, đồng thời khai thác tối đa khả năng xử lý đồng thời của Haskell nhằm đảm bảo tính tương tác liên tục và mượt mà. Trong mô hình này, client không thực hiện các tính toán phức tạp mà chủ yếu đóng vai trò cầu nối giữa người chơi và server, tiếp nhận thông tin, hiển thị trạng thái trò chơi, và truyền tải các hành động của người dùng đến server theo thời gian thực.

- Xử lý I/O đồng thời: client sử dụng *race_* để chạy song song hai luồng I/O chính trong hàm run: *serverHandler* (nhận dữ liệu mạng) và *userInputHandler* (nhận dữ liệu từ console). Điều này là bắt buộc để client có thể vừa hiển thị cập nhật từ server ngay lập tức, vừa sẵn sàng nhận lệnh từ người dùng.

- Quản lý trạng thái nội bộ: client duy trì trạng thái giao diện (*ClientState*) trong một *TVar* để đồng bộ hóa giữa hai luồng. Luồng *serverHandler* sẽ cập nhật các thông tin như *IsMyTurn* và *GamePhase* vào *TVar*. Luồng *userInputHandler* sẽ đọc các thông tin đó để biết khi nào được phép gửi lệnh *MOVE* và khi nào cần gửi *REMATCH*.

- Bảo vệ giao diện: Để tránh lỗi hiển thị chồng chéo khi cả hai luồng cố gắng ghi lên console cùng lúc (một luồng vẽ lại bàn cờ, một luồng in nhắc lệnh), client sử dụng *MVar()* làm khóa (lock) trong hàm *serverHandler*. Việc bọc toàn bộ logic cập nhật trạng thái và vẽ lại giao diện bên trong *withMVar lock* đảm bảo rằng các thao tác I/O lên màn hình luôn diễn ra một cách tuần tự và nguyên tử.

2. Giao thức giao tiếp

Hệ thống giao tiếp giữa client và server được thiết kế dựa trên mô hình chuỗi ký tự có cấu trúc dạng HEADER:DATA. Cách biểu diễn này cho phép hai bên trao đổi thông tin một cách có tổ chức và dễ dàng phân tích qua giao thức TCP Socket.

Phía server: Hàm *parseCommand* chịu trách nhiệm nhận chuỗi từ client (ví dụ: "MOVE:1,2") và chuyển nó thành kiểu dữ liệu có cấu trúc (Move (1, 2)), đảm bảo các lệnh xử lý (như *handleMove* hay *handleRematch*) luôn nhận được dữ liệu đầu vào hợp lệ. Nếu quá trình phân tích thất bại (do sai cú pháp hoặc định dạng không hợp lệ), server sẽ phản hồi lại với thông báo lỗi. **Phía client:** Hàm *serverHandler* và *parseAndDisplay* sử dụng break (==

':') để tách header (ví dụ: "TURN") khỏi data (ví dụ: "YOURS"), sau đó cập nhật trạng thái hoặc hiển thị thông báo tương ứng.

a. Tin nhắn gửi từ server tới client (S –> C)

Header	Nội dung dữ liệu	Mục đích và xử lý của client
BOARD	Chuỗi biểu diễn *Board* (*[[Maybe Player]]*)	Cập nhật giao diện: luôn đi kèm sau mỗi nước đi hợp lệ. client dùng để gọi hàm *printBoard* và kích hoạt *clearScreen* để vẽ lại toàn bộ bàn cờ.
PLAYER	X hoặc O	Khởi tạo: gán quân cờ của client, dùng để cập nhật *ClientState* và hiển thị "You are Player X/O".
STATUS	Playing, Winner X/O, Draw	Trạng thái game: client sử dụng *Winner* hoặc *Draw* để chuyển trạng thái nội bộ sang *GameOver* và hiển thị thông báo kết quả.
TURN	YOURS hoặc WAIT	Điều phối lượt đi: client sử dụng để cập nhật trường *Bool* (*IsMyTurn*) trong *ClientState*, từ đó kiểm soát việc hiển thị nhắc lệnh (> *Enter your move...*).
ERROR	Mô tả lỗi	Phản hồi lỗi: thông báo cho người chơi biết nước đi của họ không hợp lệ (ví dụ: *ERROR:Not your turn, ERROR:Invalid move*).

b. Tin nhắn gửi từ client tới server (C → S)

Header	Nội dung dữ liệu	Trạng thái gửi đi (client)	Xử lý của server
MOVE	r,c (ví dụ: 1,2)	Gửi khi *GamePhase* = *Playing*. Client đọc input và gửi chuỗi nguyên bản (ví dụ: "MOVE:1,2").	*handleMove* (kiểm tra lượt đi và tính hợp lệ trên *GameState –> TVar*).
REMATCH	Không có	Gửi khi *GamePhase* = *GameOver* và người chơi nhập "yes" hoặc "y".	*handleRematch* (sử dụng *RematchState –> TVar* và check để đồng bộ hóa).
QUIT	Không	Gửi khi GamePhase = GameOver và người chơi nhập "no" hoặc bất kỳ input không phải "yes" nào khác.	*handleRematch* (đóng phiên, kích hoạt hủy bỏ luồng đối thủ và giải phóng tài nguyên).

3. Các trạng thái trò chơi

Việc quản lý trạng thái là một trong những phần phức tạp nhất của trò chơi, để đảm bảo tính chính xác và an toàn, hệ thống sẽ phân tách rõ rệt việc quản lý trạng thái ở hai cấp độ là trạng thái logic và trạng thái giao diện.

Server sẽ quản lý logic game bằng cách định nghĩa các biến giao dịch TVar chứa các kiểu dữ liệu đại số (ADT). Trạng thái cốt lõi của ván đấu được lưu trong *GameState*, là một *TVar* chứa đồng thời bàn cờ hiện tại, lượt người chơi hiện tại, và trạng thái ván đấu (*GameStatus*).

data GameStatus = Playing | Winner Player | Draw

type GameState = TVar (Board, Player, GameStatus)

type RematchState = TVar (Maybe Bool, Maybe Bool)

Playing là trạng thái mặc định của ván đấu khi vừa bắt đầu hoặc sau một nước đi hợp lệ mà chưa phân định thắng/thua/hòa. Trong trạng thái này, hàm *handleMove* sẽ lắng nghe và xử lý các lệnh *MOVE*. Nó sẽ thực hiện kiểm tra logic (đúng lượt, ô hợp lệ) và sau mỗi nước đi hợp lệ, server sẽ gọi hàm *checkWinner* để quyết định xem trạng thái tiếp theo sẽ là *Playing*, *Winner*, hay *Draw*. Trạng thái ***Winner Player*** được kích hoạt ngay khi hàm *checkWinner* phát hiện điều kiện thắng. ***Draw*** là trạng thái tương tự được kích hoạt khi hàm *checkWinner* phát hiện tất cả các ô trên bàn cờ đã được lấp đầy mà không có ai thắng. Khi *GameState* được cập nhật sang một trong hai trạng thái này, server ngay lập tức ngừng chấp nhận logic xử lý *MOVE*. Bất kỳ lệnh *MOVE* nào nhận được từ client sẽ bị từ chối với lỗi *ERROR:Game is over...* Thay vào đó, server chuyển sang chế độ chờ đợi phản hồi *REMATCH* hoặc *QUIT* từ cả hai người chơi, được quản lý thông qua *RematchState*. *RematchState* cũng sử dụng *TVar* cho phép đồng bộ hóa việc tạm dừng luồng bằng hàm *check*, loại bỏ rủi ro về lỗi tranh chấp dữ liệu và đảm bảo tính nguyên tử cho mọi truy cập trạng thái.

Client sẽ quản lý giao diện và cách diễn giải input của người dùng bằng một ADT tương tự, được lưu trong *ClientState* (một *TVar*) để đồng bộ hóa hai luồng I/O nội bộ. *ClientState* chứa quân cờ của client, trạng thái giao diện (*GamePhase*), và một giá trị boolean cho biết có phải lượt đi của người chơi hay không (*IsMyTurn*).

data GamePhase = Playing | GameOver

type ClientState = TVar (Maybe Player, GamePhase, Bool)

Playing là trạng thái hoạt động chính của client. Khi ở trạng thái *Playing*, luồng *serverHandler* chịu trách nhiệm hiển thị đầy đủ thông tin ván đấu, bao gồm bàn cờ và trạng thái lượt đi (*Turn: YOURS/WAIT*). Lời nhắc nhập lệnh (*> Enter your move...*) chỉ xuất hiện khi client nhận được tin nhắn *TURN:YOURS* từ server. Tương ứng, luồng *userInputHandler* sẽ diễn giải mọi input từ người dùng như một lệnh *MOVE* và gửi trực tiếp đến server để chờ xác thực. Client sẽ duy trì trạng thái này cho đến khi luồng *serverHandler* bắt được một tin nhắn *STATUS:Winner...* hoặc *STATUS:Draw*, lúc đó nó sẽ tự động chuyển *ClientState* sang *GameOver*. Khi ở *GameOver*, *serverHandler* sẽ ẩn đi các thông tin không còn liên quan và

hiển thị một lời nhắc chơi lại: *Play again? (yes/no)*: Về mặt xử lý input, *userInputHandler* không còn gửi lệnh *MOVE* nữa, mà thay vào đó diễn giải input của người dùng. Các câu trả lời như "yes" hoặc "y" sẽ được chuẩn hóa thành lệnh *REMATCH*, trong khi bất kỳ input nào khác (như "no", "quit", v.v.) đều được coi là lệnh *QUIT* và gửi về server. Trạng thái *GameOver* này được duy trì cho đến khi người chơi đồng ý chơi lại (client tự reset về *Playing* và chờ server gửi bàn cờ mới), hoặc nhận được tin nhắn “Disconnecting…” từ server, khiến client kết thúc chương trình.

III. THIẾT KẾ CHƯƠNG TRÌNH

Chương này sẽ trình bày chi tiết cách thiết kế và hiện thực hóa chương trình game caro bằng ngôn ngữ Haskell, tập trung làm rõ cách các nguyên lý lập trình hàm và cơ chế quản lý đồng thời được áp dụng trong thực tế.

1. Biểu đồ module

- Module server

Server.hs

```
│
├── main
│   └── loop
│       └── gameSession (Luồng mới cho mỗi cặp đấu)
│           ├── broadcastNewState (Gửi trạng thái ban đầu)
│           ├── playerListener (Player X)
│           │   ├── parseCommand
│           │   ├── handleMove
│           │   │   ├── isValidMove
│           │   │   ├── updateBoard
│           │   │   ├── checkWinner
│           │   │   └── broadcastNewState (Cập nhật sau nước đi)
│           │   └── handleRematch (Xử lý chơi lại/thoát)
│           │       └── broadcastNewState (Cập nhật nếu chơi lại)
│           └── playerListener (Player O)
│               ├── parseCommand
│               ├── handleMove
│               │   ├── isValidMove
│               │   ├── updateBoard
│               │   ├── checkWinner
│               │   └── broadcastNewState (Cập nhật sau nước đi)
│               └── handleRematch (Xử lý chơi lại/thoát)
│                   └── broadcastNewState (Cập nhật nếu chơi lại)
└── (Network setup: resolve, open)
```

- Module client

Client.hs

```
│
├── main
│   └── run (Khởi chạy 2 luồng song song)
│       ├── serverHandler (Luồng 1)
│       │   └── parseAndDisplay
│       │       └── printBoard
```

```
│       └── userInputHandler (Luồng 2)
└── (Network setup: resolve, open)
```

2. Tính thuần túy

Hàm ***updateBoard*** minh họa nguyên tắc bất biến dữ liệu của Haskell. Hàm này không hề sửa đổi bàn cờ gốc mà thay vào đó, nó nhận *Board* cũ, tọa độ (r, c), và quân cờ (player), rồi trả về một *Board* mới hoàn toàn. Cơ chế hiện thực sử dụng các thao tác danh sách cơ bản (take, drop) để xây dựng lại hàng bị thay đổi và ghép lại toàn bộ bàn cờ. Việc tạo ra một *Board* mới sau mỗi nước đi là yếu tố then chốt hỗ trợ tính an toàn của cơ chế STM.

Hàm ***checkWinner*** là bộ não luật chơi, xác định trạng thái kết thúc ván đấu, chỉ nhận *Board* và trả về *GameStatus* một cách thuần túy. Hàm thực hiện kiểm tra điều kiện thắng 3 quân liên tiếp theo bốn hướng chính: ngang, dọc, chéo chính và chéo phụ. Nếu không tìm thấy người thắng trong bất kỳ hướng nào, hàm sẽ kiểm tra điều kiện cuối cùng là Hòa (Draw) bằng cách xác định tất cả các ô trên bàn cờ đã được lấp đầy hay chưa (*all (all isJust) board*). Sự tồn tại của các hàm thuần túy này giúp Server tập trung vào việc điều phối và đồng bộ hóa mà không cần lo ngại về việc kiểm tra lại logic game.

3. Luồng xử lý lệnh

Luồng logic trong ***playerListener*** bắt đầu bằng I/O mạng và kết thúc bằng việc điều hướng hàm xử lý phù hợp:

- Tiếp nhận I/O: luồng liên tục gọi *recv conn 1024* để nhận dữ liệu thô từ client.
- Phân tích và chuyển đổi: dữ liệu thô ngay lập tức được chuyển đổi thành kiểu dữ liệu có cấu trúc bằng hàm *parseCommand*.
- Điều hướng logic: dựa trên kết quả phân tích, luồng điều hướng đến *handleMove* (nếu đang ở trạng thái *Playing*) hoặc *handleRematch* (nếu game đã kết thúc), đảm bảo chỉ có logic thích hợp được thực thi

Hàm ***handleMove*** chứa logic kiểm tra tính hợp lệ và là nơi diễn ra thao tác thay đổi trạng thái game. Chức năng của nó minh họa việc sử dụng *TVar* trong luồng I/O. Hàm này thực hiện các bước kiểm tra logic nghiêm ngặt như kiểm tra lượt đi và tính hợp lệ của ô đánh trước khi cập nhật trạng thái. Nếu bất kỳ kiểm tra nào thất bại, server gửi ngay tin nhắn *ERROR* trở lại client đó. Chỉ khi nước đi hợp lệ, server gọi các hàm thuần túy *(updateBoard*, *checkWinner*) để tính toán trạng thái mới, sau đó thực hiện ghi nguyên tử bằng *atomically $ writeTVar gameState newState*. Sau khi trạng thái được cập nhật an toàn, server gọi hàm *broadcastNewState*. Hàm này gửi cùng một bàn cờ mới đến cả hai client, nhưng tùy chỉnh tin

nhắn điều khiển *TURN* (*YOURS* hoặc *WAIT*) cho từng người chơi, qua đó khẳng định vai trò điều phối lượt đi của server.

Hàm ***handleRematch*** chịu trách nhiệm xử lý các lệnh *REMATCH* và *QUIT* sau khi ván đấu kết thúc. Đây là nơi STM được sử dụng để đồng bộ hóa quyết định giữa hai luồng, giải quyết bài toán chờ đợi một cách hiệu quả. Luồng của người chơi sẽ cập nhật lựa chọn của họ vào *RematchState*, sau đó bị tạm dừng một cách hiệu quả bằng hàm *check* bên trong khối *atomically*. Luồng sẽ không tiêu tốn tài nguyên CPU mà chỉ được đánh thức khi đối thủ đưa ra quyết định. Khi cả hai luồng được đánh thức, server kiểm tra sự đồng thuận và nếu cả hai đồng ý, server thực hiện một giao dịch nguyên tử khác để reset cả *GameState* và *RematchState*, sau đó phát sóng bàn cờ mới để khởi động lại ván đấu.

4. Cơ chế đồng bộ hóa bằng STM

Việc sử dụng Software Transactional Memory (STM) là nền tảng kỹ thuật quan trọng nhất trong server, đảm bảo tính nguyên tử và toàn vẹn của dữ liệu trong môi trường đa luồng. STM được áp dụng trực tiếp để giải quyết hai kịch bản đồng bộ hóa kinh điển trong các hàm *handleMove* và *handleRematch*.

- Kịch bản 1: Giải quyết lỗi tranh chấp dữ liệu (*handleMove*)
  Vấn đề tranh chấp dữ liệu xảy ra khi hai luồng (X và O) cố gắng đọc và ghi vào trạng thái game chung (*GameState*) cùng một lúc. STM giải quyết vấn đề này thông qua cơ chế giao dịch nguyên tử (atomic transaction).

  Luồng xử lý bắt đầu bằng một hình thức kiểm soát lạc quan (optimistic control): server kiểm tra logic (đúng lượt đi, ô hợp lệ) trước khi thực hiện thay đổi. Nếu người chơi gửi *MOVE* không đúng lượt (*currentPlayer /= myPlayer*), lệnh đó bị từ chối với lỗi *ERROR:Not your turn*.

  Chỉ khi nước đi hợp lệ, server thực hiện thao tác ghi nguyên tử bằng lệnh *atomically $ writeTVar gameState newState*. Khối *atomically* đảm bảo giao dịch này hoặc thành công hoàn toàn, hoặc thất bại hoàn toàn. Nếu hai luồng cố gắng ghi vào *GameState* cùng lúc, STM sẽ đảm bảo chỉ có một giao dịch được cam kết thành công. Giao dịch còn lại sẽ được hủy bỏ và tự động chạy lại. Lúc này, khi luồng bị chạy lại, nó sẽ đọc được *GameState* đã được cập nhật, và logic kiểm tra lượt đi sẽ bắt lỗi, từ đó đảm bảo tính tuần tự của ván cờ.

- Kịch bản 2: Đồng bộ hóa chờ đợi (*handleRematch*)

Cơ chế chơi lại cho thấy STM hiệu quả trong việc tạm dừng luồng không tốn tài nguyên. Luồng xử lý bắt đầu bằng việc ghi nhận quyết định của người chơi vào *RematchState*. Sau đó, luồng đi vào khối *atomically* và gọi hàm *check* (*isJust pX && isJust p*O). Hàm *check* sẽ khiến luồng tạm dừng mà không tiêu tốn tài nguyên CPU (Idle). Luồng chỉ được đánh thức khi *RematchStat*e bị thay đổi (tức là khi đối thủ đưa ra quyết định) và điều kiện của *check* là *True*. Khi cả hai luồng được đánh thức, server thực hiện giao dịch cuối cùng để reset cả *GameState* và *RematchState* về trạng thái ban đầu, khởi động ván mới đồng thời cho cả hai bên. Cơ chế này loại bỏ nhu cầu sử dụng các cấu trúc khóa truyền thống cho bài toán chờ đợi

5. Quản lý tương tác trên client

Client được thiết kế để chạy hai tác vụ I/O chính đồng thời. Hàm ***run*** sử dụng *race_* để khởi chạy *serverHandler* (nhận tin nhắn mạng) và *userInputHandler* (nhận lệnh từ console) tránh tình trạng luồng bị tắc nghẽn khi chờ I/O. Để giải quyết vấn đề xung đột khi hai luồng cố gắng ghi lên console (một luồng vẽ bàn cờ, một luồng in nhắc lệnh), client còn sử dụng *MVar()* làm khóa. Hàm *serverHandler* bọc toàn bộ logic cập nhật trạng thái và vẽ lại giao diện bên trong *withMVar lock*. Cơ chế khóa này đảm bảo rằng các thao tác I/O lên màn hình luôn diễn ra tuần tự và nguyên tử, ngăn chặn lỗi hiển thị chồng chéo thường thấy trong các ứng dụng console đa luồng.

Luồng ***serverHandler*** còn là trung tâm cập nhật trạng thái của client. Nó liên tục nhận các tin nhắn server, cập nhật *ClientState* nội bộ bằng *atomically $ modifyTVar'*. Client áp dụng chiến lược "vẽ lại trên cập nhật", khi nhận tin nhắn *BOARD: [...]*, client thực hiện *clearScreen* và gọi *printBoard* để vẽ lại toàn bộ bàn cờ dựa trên dữ liệu mới nhất. Điều này đảm bảo giao diện luôn đồng bộ với trạng thái server.

Luồng ***userInputHandle***r chịu trách nhiệm đọc trạng thái nội bộ từ *ClientState* để điều chỉnh hành vi gửi lệnh. Logic này cho phép client gửi lệnh khác nhau tùy thuộc vào trạng thái game. Khi *GamePhase = Playing*, luồng diễn giải input là lệnh *MO*VE; khi *GamePhase = GameOver*, input được dịch thành *REMATCH* (nếu là "yes") hoặc *QUIT* (nếu là input khác). Đặc biệt, nếu người chơi đồng ý chơi lại, lient ngay lập tức thực hiện giao dịch *atomically* để reset trạng thái nội bộ về *Playing*, chuẩn bị cho ván đấu mới trước khi gửi lệnh *REMATCH* lên server

IV. ĐÁNH GIÁ VÀ KẾT LUẬN

1. Kết quả đạt được

Đồ án đã hiện thực thành công game caro theo kiến trúc Client-Server, đáp ứng đầy đủ các mục tiêu kỹ thuật đã đặt ra. Hệ thống chứng minh khả năng áp dụng Haskell vào lập trình mạng khi client và server giao tiếp ổn định qua socket TCP/IP.

Về mặt lập trình hàm, logic cốt lõi của game, bao gồm hàm *checkWinner* và *updateBoard*, được giữ hoàn toàn thuần túy và bất biến, tách biệt triệt để khỏi tác vụ I/O, giúp mã nguồn ổn định và dễ bảo trì. Bên cạnh đó, cơ chế STM đã được ứng dụng hiệu quả để bảo vệ trạng thái chung (*GameState, RematchState*) trên server, giải quyết thành công các vấn đề lỗi tranh chấp dữ liệu và đồng bộ hóa chờ đợi một cách an toàn.

Hệ thống cũng đạt được mục tiêu về tương tác đa luồng: client sử dụng *race_* để chạy đồng thời hai luồng I/O, mang lại trải nghiệm tương tác liền mạch trên giao diện dòng lệnh, đồng thời giải quyết được vấn đề xung đột ghi màn hình bằng *MVar*. Cuối cùng, hệ thống hỗ trợ đầy đủ chức năng logic của ván đấu, bao gồm các trạng thái kết thúc và cơ chế đồng bộ hóa chơi lại.

2. Những thách thức kỹ thuật

Thách thức kỹ thuật	Phân tích	Giải pháp
Phân tích I/O không đồng bộ	Việc phân tích chuỗi tin nhắn (*recv*) từ socket không thể đảm bảo mỗi lần nhận chỉ là một tin nhắn. Chuỗi có thể bị cắt hoặc chứa nhiều lệnh.	Sử dụng *C8.lines* (trong *serverHandler* của client) để tách chuỗi byte thành từng dòng tin nhắn riêng biệt, sau đó dùng *forM_* để xử lý tuần tự từng lệnh.

Bảo vệ giao diện console	Hai luồng client (*serverHandler* và *userInputHandle*r) cố gắng ghi lên console cùng lúc, gây ra lỗi hiển thị chồng chéo.	Áp dụng *MVar* (lock) trong *serverHandler* để tuần tự hóa việc cập nhật *ClientState* và vẽ lại giao diện.
Xử lý ngoại lệ trong đa luồng	Khi một người chơi *QUIT* hoặc ngắt kết nối đột ngột, server phải đảm bảo luồng đối thủ cũng kết thúc và tài nguyên được giải phóng.	Sử dụng *forkFinally* để đảm bảo đóng socket sau khi luồng kết thúc và *race_* để kích hoạt hủy bỏ luồng đối thủ ngay khi một luồng thoát bằng *E.throwIO*.
Đồng bộ hóa lượt đi	Đảm bảo tính tuần tự nghiêm ngặt của lượt đi chỉ bằng STM.	Kết hợp logic kiểm tra (if *currentPlayer /= myPlayer*) với cơ chế *retry* tự động của STM. Điều này đảm bảo rằng trạng thái chỉ được ghi nếu logic ban đầu là chính xác.

3. Hướng phát triển

Để mở rộng và hoàn thiện hệ thống, có ba hướng phát triển kỹ thuật chính nên được ưu tiên. Trước hết là việc mở rộng logic game bằng cách nâng cấp hàm *checkWinner* và các logic liên quan để hỗ trợ bàn cờ 15x15 và luật Gomoku 5 quân chuẩn, đòi hỏi tối ưu hóa các thuật toán kiểm tra. Thứ hai là tăng cường khả năng quản lý hệ thống bằng cách hiện thực một luồng "Lobby Manager". Luồng này sẽ sử dụng các cấu trúc đồng thời như TVar [GameSession] để quản lý danh sách các trận đấu đang diễn ra và cho phép người chơi tìm kiếm và tham gia nhiều trận đấu song song. Cuối cùng, hướng phát triển quan trọng nhất là cải thiện trải nghiệm người dùng bằng cách thay thế giao diện dòng lệnh bằng một giao diện đồ họa, có thể thông qua các thư viện như Gloss hoặc Threepenny-GUI, nhằm mang lại tương tác trực quan và hiện đại hơn.

V. HƯỚNG DẪN CÀI ĐẶT VÀ VẬN HÀNH

1. Chuẩn bị môi trường và tải dự án

Đầu tiên, người dùng cần chuẩn bị các môi trường sau:

- Git: cài đặt git để sao chép mã nguồn từ repository. Mã nguồn của đồ án: **https://github.com/fuji005/Final-project-haskell-caro-game.git**
- Haskell Tool Stack (GHC): cần cài đặt trình biên dịch Haskell (GHC) và công cụ quản lý dự án cabal. Cabal được sử dụng để quản lý các thư viện phụ thuộc (network, stm, async).
- Hệ điều hành: Linux, macOS, hoặc Windows (cần sử dụng command Prompt/PowerShell hoặc terminal có hỗ trợ lệnh ANSI để hiển thị bàn cờ)..

Tiếp theo, để vận hành trò chơi cần xây dựng dự án theo các bước sau:

- Tải mã nguồn bằng cách mở terminal và sử dụng lệnh "git clone" để tải dự án về máy

git clone https://github.com/fuji005/Final-project-haskell-caro-game.git

- Di chuyển vào thư mục dự án:

cd Final-project-haskell-caro-game\Caro game haskell

- Chạy các lệnh sau để cabal cài đặt các thư viện cần thiết và biên dịch dự án:

cabal update

cabal build

2. Vận hành trò chơi

Trò chơi cần được chạy theo thứ tự: server → client X → client O

- Khởi động server

- Mở terminal 1 (server): trong thư mục dự án, chạy file thực thi của Server bằng lệnh:

cabal run server

- Xác nhận: server sẽ hiển thị thông báo *Server is listening on port 3000...* và *Waiting for 2 players to connect....*

- Khởi động clients và bắt đầu ván đấu

- Mở terminal 2 (client X): chạy client đầu tiên bằng lệnh:

cabal run client

- Terminal 2 sẽ hiển thị *Connected* và *You are Player X*, sau đó chờ đối thủ.

- Mở Terminal 3 (client O): chạy client thứ hai bằng lệnh:

cabal run client

- Bắt đầu ván đấu: ngay khi client O kết nối, server sẽ khởi tạo luồng game. Cả hai client sẽ nhận được thông báo về bàn cờ trống và client X sẽ nhận được nhắc lệnh: > Enter your move (e.g., MOVE:1,1):.

3. Quy trình chơi và giao tiếp

- Thực hiện nước đi: người chơi nhập lệnh theo cú pháp MOVE:hàng,cột (ví dụ: MOVE:1,1 để đánh vào hàng 1, cột 1) và nhấn Enter.

- Cập nhật trạng thái: sau mỗi nước đi hợp lệ, màn hình của cả hai client sẽ được xóa và vẽ lại với bàn cờ mới nhất, và lượt đi sẽ được chuyển đổi.

- Chức năng chơi lại: khi game kết thúc, cả hai client sẽ nhận được lời nhắc *Play again? (yes/no):*.
- Nhập "yes" (hoặc y) để gửi lệnh *REMATCH*.
- Nhập "no" (hoặc bất kỳ ký tự nào khác) để gửi lệnh *QUIT*.
- Nếu cả hai đồng ý *REMATCH*, server sẽ reset trạng thái và ván mới bắt đầu. Nếu một bên chọn *QUIT*, cả hai client sẽ nhận thông báo ngắt kết nối và thoát chương trình.