CS 214 Spring 2023
Project III: Tic-Tac-Toe On-line

David Menendez

Due: April 25, 2023, at 11:59 PM (ET)

For this project, you and your partner will design and implement an on-line multiplayer Tic-Tac-Toe game service. You will write two programs `ttt`, which implements a simple game client, and `ttts`, the server used to coordinate games and enforce the rules.

1 Tic-Tac-Toe

The game Tic-Tac-Toe is played on a 3×3 grid by two players, designated X and O. Players alternately mark empty grid cells with their symbol until the game ends. Ending conditions are:

- One player wins by placing 3 marks in a single row, column, or diagonal.
- Both players draw by filling the board without either player winning.
- One player resigns.
- Both players agree to draw.

We arbitrarily choose X to play first.

2 Software

The role of `ttt` will be to connect to the service (provided `ttts`), display the current state of the grid to the player, report moves by the other player, and obtain and transmit moves made by its player.

The role of `ttts` will be to pair up players, choose who will go first, receive commands from the players, and track the state of the grid. In particular, `ttts` will ensure that invalid moves are rejected and determine when the game has ended.

The arguments to `ttt` are the domain name and port number of the desired service. The argument to `ttts` is the port number it will use for connection requests.

3 Protocol

The game protocol involves nine message types:

- PLAY *name*
- WAIT
- BEGN *role name*
- MOVE *role position*
- MOVD *role position board*
- INVL *reason*
- RSGN
- DRAW *message*
- OVER *outcome reason*

Figure 1 shows an example of communication between the server and two clients during the set-up and first two moves of a game. Figure 2 shows two proposed draws, the first rejected and the second accepted.

3.1 Message format

Messages are broken into fields. A field consists of one or more characters, followed by a vertical bar. The first field is always a four-character code. The second field gives the length of the remaining message in bytes, represented as string containing a decimal integer in the range 0–255. This length does not include the bar following the size field, so a message with no additional fields, such as WAIT, will give its size as 0. (Note that this documentation will generally omit the length field when discussing messages for simplicity.) Subsequent fields are variable-length strings.

Note that a message will always end with a vertical bar. Implementations must use this to detect improperly formatted messages.

The field types are:

name Arbitrary text representing a player's name.

role Either X or O.

position Two integers (1, 2, or 3) separated by a comma.

board Nine characters representing the current state of the grid, using a period (.) for unclaimed grid cells, and X or O for claimed cells.

reason Arbitrary text giving why a move was rejected or the game has ended.

message One of S (suggest), A (accept), or R (reject).

outcome One of W (win), L (loss), or D (draw).

Figure 3 gives some example messages.

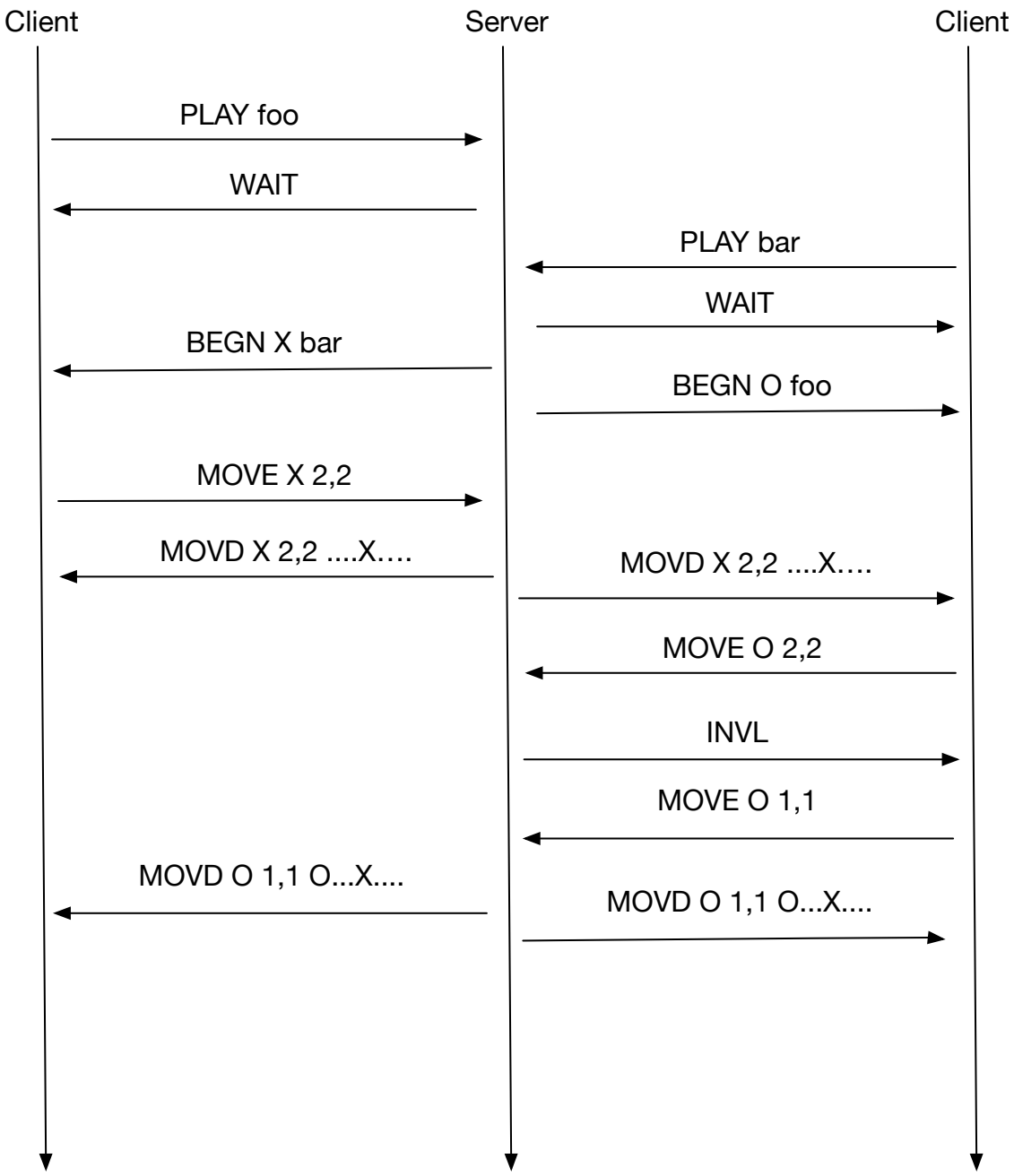

Figure 1: Message flow showing game setup and two moves

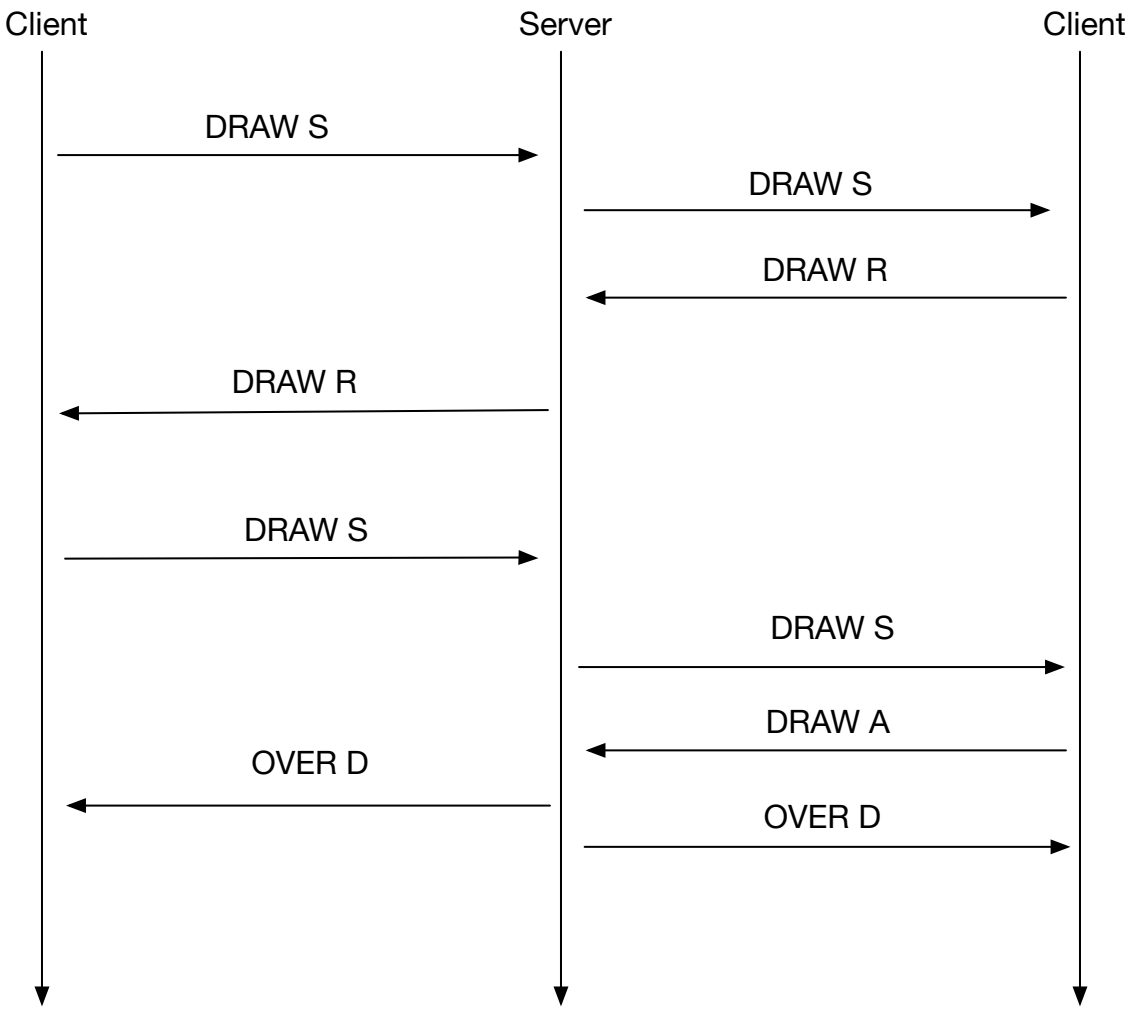

Figure 2: Message flow showing rejected and accepted draws

```
NAME|10|Joe Smith|
WAIT|0|
MOVE|6|X|2,2|
MOVD|16|X|2,2|....X....|
INVL|24|That space is occupied.|
DRAW|2|S|
OVER|26|W|Joe Smith has resigned.|
```

Figure 3: Examples of messages

3.2 Messages sent by client

PLAY Sent once a connection is established. The third field gives the name of the player.

The expected response from the server is WAIT. The server will respond INVL if the name cannot be used (e.g., is too long).

MOVE Indicates a move made by a player. The third field is the player's role and the fourth field is the grid cell they are claiming.

The server will respond with MOVD if the move is accepted or INVL if the move is not allowed.

RSGN Indicates that the player has resigned.

The server will respond with OVER.

DRAW Depending on the *message*, this indicates that the player is suggesting a draw (S), or is accepting (A) or rejecting (R) a draw proposed by their opponent.

Note that DRAW A or DRAW R can only be sent in response to receiving a DRAW S from the server.

3.3 Messages sent by server

WAIT Sent in response to the PLAY message.

BEGN Indicates that play is ready to begin. The third field is the role (X or O) assigned to the player receiving the message and the fourth field is the name of their opponent.

If the role is X, the client will respond with MOVE, RSNG, or DRAW. Otherwise, the client will wait for MOVD.

MOVD Sent to both participants to indicate that a move has occurred. The third field is the role of the player making the move and the fourth field gives the current state of the grid.

The player that made the move will wait for their opponent's move. The opponent client will respond with MOVE, RSGN, or DRAW.

INVL Indicates that the client's message was invalid. The third field is arbitrary text explaining why the message was rejected. INVL is both to reject illegal moves and to report protocol errors. When used to reject a protocol error (such as a message being sent at an inappropriate time), the explanation must begin with an exclamation point (!).

DRAW If one player suggests a draw, the server will send a draw suggestion to their opponent. The expected response is DRAW A or DRAW R.

After receiving DRAW A, the server ends the game and sends OVER to both clients.

After receiving DRAW R, the server sends DRAW R to the player that had sent DRAW S.

OVER Indicates that the game has ended. The second field indicates whether the recipient has won (W), lost (L), or drawn (D). The third field is arbitrary text explaining the outcome (e.g., one player has completed a line, someone has resigned, the grid is full).

3.4 Game setup

Connections are initiated by the client, which sends PLAY. In response, the server will reply with WAIT or INVL. Clients will wait for the next message.

Once the server has connected two players, it will send BEGN to both players, along with their role and their opponent's name. Player X will move first.

3.5 Game move

Moving player The client for the player who is moving will send MOVE, RSGN, or DRAW S.

In response to MOVE, the server will reply MOVD or INVL. A response of MOVD means the move was accepted and it is now the opposing player's turn.

In response to RSGN, the server will reply OVER.

In response to DRAW S, the server will reply with DRAW R if the opposing player chooses not to draw, or OVER if the opposing player has chosen to draw.

Waiting player The client for the player who is not moving will wait for the server to indicate the opposing player's move, which will be one of MOVD, DRAW S, or OVER.

After receiving MOVD, it is now that player's turn.

After receiving DRAW S, the player can respond with DRAW R or DRAW A and then wait for the next move.

After receiving OVER, the game has ended.

4 Implementation

Your implementation of `ttts` will accept as an argument the port number it will use to listen for incoming connections. As player clients connect, it will pair them into games, assign roles to each player, and maintain the state of the game grid until the game has ended. Doing so will involve at least three sockets: one for listening, and one for each active connection.

It is recommended that `ttts` write a log to STDOUT, noting incoming connections and moves as they are made.

4.1 Single game (70 points)

For partial credit, design your server to manage at most one active game at a time. While one game is in session, no additional player clients will be able to register. The base protocol is designed so that the server can always predict which client will send the next message, meaning your implementation can use a single thread to manage all three sockets.

4.2 Concurrent games (100 points)

For full credit, `ttts` must be able to handle multiple concurrent games. That is, `ttts` must be able to listen for incoming connection requests while also managing one or more active games. This will require some degree of multitasking (recommended) or the use of `select()`.

As an additional requirement, `ttts` must track the names of all players currently playing, and require all players to have unique names. That is, a PLAY message including a name that matches

a name of a player in an on-going game will receive INVL with the explanation "name already in use". Your implementation will need to use locks to ensure that access to the shared list of in-use names is performed safely. Describe your use of locks in your README.

The design of the game protocol means it is possible (but not required) to handle both connections for a single game with one thread.

4.3 Concurrent games with interruption (120 points)

For extra credit, extend the protocol to allow players to resign or request a draw when it is not their turn.

Your implementation will need to wait for incoming messages on both connections simultaneously, and cancel blocked calls to `read()` when necessary. If using multiple threads to manage a game (recommended), `ttts` will need to use locks to protect the integrity of the game state.

One way to interrupt a blocked system call is to send a signal to the blocked thread, assuming the thread has a signal handler set up appropriately. The function `pthread_kill()` can be used to send a signal to a single thread, and `pthread_sigmask()` can be used to control which threads can receive specific signals. The signals SIGUSR1 and SIGUSR2 may be used for this purpose, as they will never be sent by the OS.

5 Submission

A system will be provided for you to declare your partnership prior to submission, details forthcoming. Determine in advance which partner will be responsible for submitting the completed assignment. Communication and coordination is key to a successful project!

Submit a Tar archive containing:

- Your source code file(s), including testing code
- Your make file
- Your README and test plan
- Any test inputs used by your testing process

Your README should be a plain text document containing the names and NetIDs of both partners and an indication of which implementation choice you made.

Your test plan should be part of the README or a separate PDF document describing how your testing strategy and test cases. Describe the scenarios you considered it important to check and how you performed those checks. Note that having a good test suite is an excellent way to detect errors that have been introduced or reintroduced into your code.

Your Tar archive should place all necessary files in the root of the archive, without use of a `src` or similar subdirectory. For example:

```
$ tar -vzcf p3.tar Makefile ttts.c ttt.c protocol.h protocol.c
```

It should be possible to extract your Tar archive (using `tar -xf p3.tar`) followed immediately by `make` in the same directory to build your `ttts`.