Lambdarinth

Andres Löh

Difficulty: high

1 The game

In the beginning, the game to implement will be described. The game, called "Lambdarinth", is – apart from minor variations – an instance of the board game "Ricochet Robots" (designed by Alex Randolph and published by various companies).

The game is played on a rectangular board (in the original, always 16x16) with an unlimited number of players (although, for practical reasons, at least one player is good to have). Between some of the squares and surrounding the whole board are walls. A number of Lambdas (in the original, always 4) of different colors are placed randomly on the board.

A problem is then posed to all the players, namely to reach a certain square with a specific Lambda. In order to solve the problem, any Lambda may be moved both horizontally and vertically, but only until it is blocked either by a wall or another Lambda. Players should try to get to the target square in as few moves as possible. However, players can just look at the board, not perform any actual moves (yet). Once they know a solution, they can shout out the number of moves their solution has. From then on, there is a timeout of typically 60 seconds. Within this time, other players can announce that they have also found solutions by shouting other numbers. Players who have already announced a solution can also improve their solutions by announcing lower numbers.

After the timeout passes, the winner is determined: The player with the lowest number who voiced the solution before any other players with the same number first gets the chance to demonstrate the solution on the board. If the player cannot show a valid solution (or the solution has too many moves), the player with the next-best solution gets a chance. If none of the players can demonstrate a solution, the game remains without a winner.

2 The task

The overall description of the task is simple. You should implement both a server and a client for the game just described. All code should be written in Haskell. You may use libraries and tools that are available on Hackage.

Client and server should communicate with each other via a network socket, on a configurable port, making use of a textual protocol that is specified in Section 3. The protocol also makes the basic structure of the game explicit.

In Section 4, additional requirements for the server and client are specified.

Finally, in Section 5, a number of ideas for extensions and improvements are given. You should try to implement a number of improvements on the initial requirements. Let your imagination go wild. There are basically no limitations, except that your client and server should still provide a mode in which they will be interchangeable with other clients and servers that implement the game protocol.

2.1 Demo server

I will try to run a demo server on the host `shell.students.cs.uu.nl`, port `7890`. You can connect to this server in order to test your own clients. The server should, in principle, implement the game and the game protocol as specified in this document, with the exception of board generation and the rejection of too simple problems as described in Section 4.2. If you think it does not, please let me know. I may update the server during the block for fixes, improvements or extensions. I will then tell you so. The server may also be temporarily unavailable if problems show up, so do not rely on its presence too much. Nevertheless, if you find the server down, let me know so that I can try to bring it up again.

Since the game protocol is textual, you can also use `telnet` to connect to the server and enter commands textually. From a Unix machine within the student network, the command

```
telnet shell.students.cs.uu.nl 7890
```

should connect to the server.

3 The protocol

Messages sent between the servers and the clients all have a common form. Each message is a single line terminated with a period immediately before the end of the line. The first word in the message identifies the kind of message. We call this word the *command*.

Since the protocol might be extended, or because incorrect clients might connect to the server, the server is required to be very robust. Whenever a message (i.e., input line) is received that is not in the required format, the message should just be discarded and not further affect the state of the server.

Clients should also be tolerant and just ignore messages they do not understand.

Clients and server make use of a different set of commands. Here is a list of all server and client commands that should be understood.

3.1 Phases of the game

We can distinguish three phases the server can be in: **Before**, **In**, or **After** a game.

Before a game the board and the positions of the Lambdas are already known, but not the problem (i.e., the target square). Players can confirm their interest in playing the following game in this phase.

After a certain time, or when all players around have confirmed, the game starts. The confirmed players participate, the others just watch. While **In** the game, players who are participating can make a bid by providing a number of moves in which they believe they can solve the problem. All bids are collected by the server, but bids with fewer moves are better than bids with more moves. If several bids have the same number of moves, the one first given counts. If several bids are provided by the same player, only the lowest counts. The game takes some amount of time. After the first bid, the remaining time is set to 60 seconds regardless of what it has been before. When this timeout has passed, the game ends.

After the game the server tries to determine a winner. In the order of the bids received, it asks the players who submitted the bids to submit a solution. There is a timeout for that as well. If a player who is asked manages to submit a correct solution in time, the server announces a winner and moves to the **Before** phase of the next game. If no player can submit a correct solution, the server announces that there have been no winners and also advances to the **Before** phase of the next game.

3.2 Client commands

`Confirm` *Player*.

The client confirms that *Player* wants to participate in the upcoming game. The server answers with either a `NameTaken` or an `AcceptedPlayer` accepted for the same *Player* if it is in the **Before** phase. In other phases, the server will just ignore a `Confirm` command. Multiple `Confirm` commands in the **Before** phase of a game by a single client are possible, so a player can choose a new name as long as the game has not started.

`Bid` *Moves*`.`

The client states that it can provide a solution to the given problem in a maximum of *Moves* moves. The server ignores this command if it is not in the **In** phase. In the **In** phase, if the bid is currently the best bid of the player in question, the server reacts with a `ReceivedBid` command.

`Done.`

The client states that it has no interest anymore in the current game and would not mind for it to end. The server ignores this command if it is not in the **In** phase. If, while in the **In** phase, all confirmed players send a `Done` command, the server can accelerate the game and immediately switch to the **After** phase.

`Solution` *Solution*`.`

The client submits a solution *Solution* to the current problem. The server ignores this command if it is not in the **After** phase and has sent a `RequestSolution` command for that player before. The server also ignores the command if the solution submitted is illegal or has too many moves. If the solution was ok, the server answers with a `Winner` command.

3.3 Server commands

`Game` *Board*.

The server sends the board and the positions of the Lambdas *Board* for the next game. The server sends this command to all clients when it starts a **Before** phase, or to a new client when it connects and the server is in the **Before** or **In** phase.

`NameTaken` *Player*.

The server rejects a `Confirm` command previously sent to the server. The `NameTaken` command indicates that the chosen name *Player* is already in use by *another* player. It should not be sent in any other situation. The server sends this command as a possible reply to a `Confirm` command only to the client that sent the `Confirm` command.

`AcceptedPlayer` *PlayerId Player*.

The server announces that a player *Player* will join the upcoming game. The *PlayerId* associated with the player is uniquely determined by the client. The *PlayerId* can be used by clients to identify if the player is a new player or has just chosen a new name. The server sends this command as a possible reaction to a `Confirm` command to all connected clients.

`PlayerGone` *Player*.

The server announces to all connected clients that a player *Player* has left the game. The server uses this command if the connection to the client of a confirmed player has been lost, regardless of the phase the server is in.

`Problem` *Players Prob Timeout*.

The server announces to all connected clients a problem *Prob* on the current board. It also gives the *Players* that will play this game (i.e., that have confirmed before the game started and not left in the meantime). The *Timeout* in seconds indicates how much time there is at most to come up with a bid. After sending this command, the server is in the **In** phase.

`ReceivedBid` *Player Moves Timeout*.

The server announces to all connected clients that it has received a bid by player *Player*, to provide a solution in at most *Moves* moves. The remaining time in the game in seconds is indicated by *Timeout*. The server sends this command as a reaction to a `Bid` command, but only if the server is in the **In** phase, and if the bid it has received is the best bid the player in question has made in the current game so far.

`RequestSolution` *Player Moves Timeout* `.`

The server announces to all connected clients that it is waiting for the submission of a valid solution with maximum number of *Moves* moves from player *Player*. It will accept this solution within *Timeout* moves. With this command, the server also announces it is in the **After** phase. While all players receive this command, only the player mentioned is requested to submit a solution using the `Solution` command.

`Winner` *Winner* `.`

The server announces to all connected clients the winner of the current game and the solution that won, or that there has been no winner. In any case, the `Winner` command marks the end of the **After** phase and the start of a new **Before** phase. The server will go on to immediately send a new *Board* command to all connected clients.

3.4 Context-free grammar

In this part, the syntax of the command arguments is specified.

Board ::= *Pos Walls Lambdas*
Walls ::= `[ ]` | `[` (*Wall* `,`)* *Wall* `]`
Wall ::= `(` *Pos* `,` *Dir* `)`
Lambdas ::= `[ ]` | `[` (*Lambda* `,`)* *Lambda* `]`
Lambda ::= `(` *Color* `,` *Pos* `)`
Moves ::= *Nat*
PlayerId ::= *Nat*
Player ::= *String*
Players ::= `[ ]` | `[` (*Player* `,`)* *Player* `]`
Prob ::= *Color Pos*
Solution ::= `[ ]` | `[` (*Step* `,`)* *Step* `]`
Step ::= `(` *Color* `,` *Dir* `)`
Timeout ::= *Nat*
Winner ::= `None` | *Player Solution*
Color ::= `Red` | `Green` | `Blue` | `Yellow`
Dir ::= `N` | `E` | `S` | `W`
Pos ::= `(` *Nat* `,` *Nat* `)`

A *Board* is specified by its size (in the form of a coordinate pair *Pos*), a list of *Walls* and a list of *Lambdas*. Lists are in Haskell syntax, surrounded by square brackets, the items separated by commas. Pairs are also in Haskell syntax, surrounded by round brackets and the two components separated by a comma.

A *Wall* is a pair of a *Pos* and a *Dir*. It indicates that moving from position *Pos* in direction *Dir* is blocked. Note that this allows for boards to have walls that only block movement in one direction. You may want to prepare your client for that possibility, and have

it show unidirectional walls differently from normal, bidirectional walls. The walls surrounding the whole board are implicitly present and may be contained in the list of walls, but do not have to be.

A *Lambda* is a pair of a *Color* and a *Pos*, indicating that the Lambda of that color is of that position. In the list of Lambdas, the colors that occur should all be different, but not all problems have to use all available colors.

A specification of *Moves* is just a natural number *Nat*.

A *PlayerId* is a natural number *Nat*. Player names *Player* are given as a *String*. Finally, *Players* are a list of *Player*.

A *Prob* is given by a *Color* followed by a *Pos* as well, indicating that the Lambda of the color *Color* has to be moved to position *Pos*. Obviously, the server should only pose problems for a Lambda color that exists on the current board.

A *Solution* is given by a list of *Step*s. A *Step* is a pair of a *Color* and a *Dir*. This means that the Lambda of the appropriate color has to be moved in direction *Dir*. A solution is valid if after performing the steps in order, starting from the initial positions of the Lambdas on the board, the problem is solved, i.e., the Lambda of the specified color is on the specified square. The length of a solution is given by the length of the list of steps.

A *Timeout* is a natural number *Nat*, specifying how many seconds are left in the game.

For *Winner*, there are two alternatives. The terminal `None` indicates that there is no winner. Otherwise, the winning *Player* and the winning *Solution* are given.

All tokens in the above grammar may be separated by spaces (not spread across many lines though, as each message takes exactly one line).

3.5 Lexical syntax

Natural numbers *Nat* are a non-empty sequence of digits. Strings *String* are sequences of characters surrounded by double quotes. In its simplest form, strings may only contain 7-bit ASCII characters and must not contain either backslashes or double quotes. Haskell-like backslash-escaped characters may be added.

4 Additional requirements

In this part, several additional requirements of both the client and the server component are given.

4.1 The client

The server and port the client tries to connect to should be configurable (via command line, preferably).

The client should keep track of the players currently in the game. This information is only available by following the announcements of the server in the `AcceptedPlayer`, `PlayerGone` and `Problem` commands.

During the game, the client should keep track of the bids that have been placed by the players, by keeping track of the `ReceivedBid` announcements. At the very least, it should be obvious to the player at any point of the game who has placed the best bid so far.

The client should provide a graphical representation of the board. The client should not rely on the board always having the same, or a specific, size. The client should provide a way to edit and submit a solution while playing the game, and it should provide a way to display the winning solution after it has been announced using `Winner`.

The client should make gaming convenient. Make it easy to place a bid even before entering a solution. Placing a bid fast is important to win the game, so it should not be complicated to do so.

Make it easy to pick a name for the player and use the same name later on in subsequent games.

4.2 The server

The server should continuously stick to the **Before**, **In**, **After** loop. It should always accept new connections, and should handle each connection in its own thread. The port on which the server listens for incoming connections should be configurable (via command line, preferably).

The server has a certain amount of flexbility in the generation of timeouts: there should be a point where a server switches from **Before** to **In** phase, even if not all players currently connected have confirmed. Non-responsive players shouldn't be able to spoil the fun for all the others. One option is to have the first `Confirm` start a timeout of, say, one minute during which other players can confirm. If all connected players confirm earlier, the game can start earlier.

Also, the server has to generate a timeout for a game initially. If a puzzle turns out too hard to solve, or if players all lose interest and quit the game or become non-responsive, the server should not stay in the game forever (during this time, other newly connected players will have to wait, after all). The demo server uses 300 seconds for this purpose, but you can choose another value.

After the first bid – at least in the standard version – the timeout should be reset to 60, and even if it previously has been less than 60. From then on, the timeout is unaffected by further bids.

The server should provide some variation in the boards and the problems. The demo server currently uses only one board, but many problems on that board. You can start with that board, which is one of the board game boards – it works well. But try to also allow a few different boards. Think about the characteristics that make a board well-suited for the game and explain/motivate your choices.

When generating problems, pay attention to the fact that the problem should be solvable. Not all positions on the board may be reachable. In particular, positions too far away from walls or even corners may be very difficult to reach.

The server should not pose problems that are solvable in less than four moves with a single robot. The demo server currently does not have this feature.

5 Potential extensions

Once you have a client and a server that adhere to the minimal requirements above, you can start implementing extensions. Here are a few ideas, but you should not feel limited by them.

- Implement an automatic or semi-automatic client. Either try to write a fully automatic client that solves problems without user interaction and tries to be competitive with interactive clients (or ideally is much faster). Alternatively, you can at least try to solve easy problems automatically, or support the user with hints.
- Different numbers of Lambdas. Instead of a maximum of four, allow five, or many more. Vary the number of make it configurable.
- Allow problems where *any* Lambda may be moved to the target square.
- Implement random board generation. Try to generate good boards and problems that are neither too easy nor too difficult.
- More board features (really use unidirectional walls, add fields that change the direction of (some) Lambdas, that shift Lambdas one position to the side, that teleport robots to some other place etc.).
- Change the physics (for instance, if a Lambda hits another Lambda, the impulse from the first carries over to the second).
- Change the scoring system (for instance, in order to encourage solutions that use many Lambdas, parallel moves of different Lambdas could just count as a single move).

- Implement more variants of the game, or even completely different, but somewhat similar games. Make the game variant choosable by the clients, or configurable when starting the server.
- Make the server more verbose and responsive. Instead of just ignoring commands it does not understand or that come in the wrong phase of the game, it could respond with appropriate informative messages about what is going on or what it is expecting.
- Make it possible for players logged into the game to communicate (chat) with each other.
- Allow multiple games on the same server being carried out in parallel (i.e., have multiple game rooms).
- Collect statistics about won or lost games in the server and make those statistics available by querying the server. Player data should be stored in a text file or a database.