Technical Documentation

# Program Flow and User Interface

|  |  |
| --- | --- |
|  | The client opens with a login interface. The user can enter a username and password. The password will be hidden by the dot character.  The user can also register a new account with the server by using the Register link. |
|  | Clicking the register link will open a new form which allows the user to enter new login credentials. There are several checks on the username and password to confirm they are suitable before the user will be allowed to register (See “Logging In and Registering” for details). |
|  | After registering an account, the register form will automatically close and the login form will refocus. Pressing Login will initiate the verification process (See “Logging In and Registering” for details). |
|  | This is the main window of the system. The window has several features. The selection boxes and number selectors on the left allow the user to select the parameters of the maze. Filling in all boxes whilst being connected to the server will unlock the Request Maze button which will send the information to the server where a maze will be generated (See “Maze Generation” for more detail.) Alternatively, by pressing Get Mazes, the selection box below the button will be populated with mazes the user has saved to the server. The user can load or delete the maze from the server with the relevant buttons. The panel on the right hosts dynamically generated charts showing local and global statistics for best times and mazes generated. |
|  | Here is the main form populated with data. The Request Maze button is now unlocked since each parameter has been filled in. Clicking the button will send the parameters to the server and a maze will be passed back to the client, being displayed in a new form. |
|  | Here is the Display form. The user can use WASD controls or the onscreen buttons to move the blue square through the maze. If they do this, a timer will start and be displayed, stopping when they reach the end of the maze. The Close button will then be unlocked, allowing access to the parameter form. The user can also use the selection box to select the algorithm they would like to request a server solve with. Pressing request solve will send the maze to the server which will send a solution back to the client before displaying the solution. The user also has the option to name their maze for a server save, or save it locally, which will open a File Explorer window where they can choose where to save the formatted image. |
|  | Here is the display form after requesting a solve using the Maze Routing algorithm. The solution is displayed with a purple line. The user can no longer attempt a manual solve. The close button has been enabled. Closing the window will refocus the parameter window and allow the user to begin the process again. |
|  | In the background, the server has been open the entire time. It logs all requests made to it in the console window. The client has exception handling on all server requests, since if the server closes unpredictably, there will be an unavoidable error which needs to be appropriately handled. (See “Network Exception Handling” for more details). |

# Project Structure

|  |  |
| --- | --- |
| Client | Server |
|  |  |
| The client’s code is mainly held within its form codebehind files. The Maze and Maze-derivative files it has contain only properties and [JsonConstructor] tagged constructors since they only need to build objects from the property structure to be read and displayed. This keeps the client lightweight. The client has a copy of every protocol file so it can send and receive the appropriate objects across the network. The client also makes use of a static class Globals (See “Global Variables” for more details.) | The server has many features in its extensive solution. It contains a folder of every protocol file so it can communicate correctly with clients. Each of these protocols have a dedicated service script in the Services folder, defining the management of the data in these requests, and what is sent back to the client. It also contains both the launchSettings.json and the appSettings.json files. These define how the server behaves (such as what port it operates across, whether it logs messages, what network protocol it uses, etc). It also contains the full definitions of all Maze classes and has the Solver and Solver-derivative classes, unlike the client. The server does not make use of global variables. |

# Techniques Used

## Network Exception Handling





Here is an example of how I have managed unpredictable server errors. When we make a network request, we can attach a deadline to it by providing a time: I have provided the current UCT time + 3 seconds for all deadlines, essentially forcing an exception after 3 seconds of server inactivity. Since we are turning an unpredictable server error into a predictable and specific error, we can catch that specific error by checking for the DeadlineExceeded status code, which is thrown when the deadline for a server request is exceeded. In the catch block, I have a procedure which is called whenever there is a server error that outputs an appropriate error message to a label on the form, as well as locking up buttons that interact with the server until a connection is re-established.

## Multithreading Exceptions

The code in my forms has many async and multithreaded methods. In a multithreaded form, threads cannot access objects from other threads without invoking a subroutine that interacts with it by proxy. However, the thread does not know when an object has been disposed, such as when a form closes, so an unpredictable error is thrown when trying to invoke access to a disposed object. Since this error causes no issue if it is ignored, we can stop a client crash if we specifically catch the ObjectDisposedException without handling it.

## Global Variables

See [PAGE] for the Globals.cs code.

Since excessive use of global variables makes code less robust, I have limited my usage to constant variables only. Furthermore, all global variables are prefixed with “g\_” to differentiate them from local variables. The only 2 non-constant globals are the username and userID, since these cannot be defined in the code as the system has many users. These are updated to the correct values at login and only read from afterwards. The version string is used to update the name of each form with the current client version. It is purely aesthetic but could have use in managing client updates. The cellWidth and cellHeight variables store the pixel width and height of cells in the maze, so the dynamically sized display form can autosize correctly. The keysize and iterations variables are used in hashing passwords (see “Logging In and Registering” for more details).

# Server Protocols

## Proto Files

See [PAGE] for the protocol files.

Protocol files end in the extension .proto and have their own language and syntax to define a service which operates across the network and the related request and response objects the service uses. The files have a set structure: first, the syntax is set. All my protocols are written in the proto3 syntax, so the top line of every file will set this. Next, the namespace the protocol will be implemented in is set. My project uses the Server namespace and Client namespace, so all my protocols are passed the Server namespace. Next, the service is defined. The service can contain many operations, represented by the rpc keyword. The operation is made up of the method name, the request object name, and the reply object name. Once all the operations have been defined in the service, the request and reply objects must be defined with what data they carry. This is done using the message keyword followed by the object name. Within the braces each variable is defined using a simple data type and an identifier. The number assigned to them is the order in which they must be passed to the collection when creating a request. Some of my protocols do not need to return a reply message to the client since they happen discreetly, such as incrementing stat values. In this case, you can import the google/protobuf/empty.proto, which when set as the reply object in the rpc definition, allows you to create a void network service. Once these protocols are defined, they must be integrated into the project to allow the service code to be written (See “Project File References” for more details).

## Project File References

See [PAGE] for the client and server project files.

Since protocol files generate obfuscated code to facilitate the low-level transfer of data across the internet, they need to be referenced in the project file, so the compiler knows to create these files when the project is built. This is done by including the path to the file in a protobuf element within an itemgroup in the project file.

# Algorithms

## Database Management

### General information

Across all SQL commands, I have implemented the same techniques.

* I have used the using keyword on all SQLite objects, so they are properly disposed after use, preventing memory leaks.
* I have used the @ symbol before all SQL commands to make them into multiline verbatim string literals. This allows me to put each component of the SQL command onto a new line, vastly increasing readability.
* I have used the tag system to manually add parameters rather than interpolating them in. This helps prevent SQL injection attacks.

### Table creation

See [PAGE] for the table creation scripts.

I have used the pragma command to turn foreign key constraints on. This means that SQL commands that would cause the primary and foreign keys across 2 tables to become mismatched are ignored.

The GlobalStats table records 3 pieces of information for each time: an integer time in milliseconds, a string display time, and the username of the user who set it. The latter 2 are used in displaying the best times on the client. The millisecond time is used in a serverside algorithm which orders the times, which would be made harder if the times were formatted strings instead.

### Triggers

See [PAGE] for the trigger body.

My database uses 1 trigger: CreateStatsRecord. The trigger creates a record in the stats table when a user registers a new account and fills it with default data. It also sets the userID of the record to be the userID of the new user. This is necessary since the stats record is not otherwise automatically created when the user registers, so many of the stat related services would break.

The code below this is similar: we need exactly 1 global stat record, so the SQL checks if there are any records in the GlobalStats table, and only adds a record if there is one. This record is filled with default data when it is initialized.

## Maze Generation

See [PAGE] for the standard implementation of InitMaze.

The InitMaze algorithm is called first to prepare the array of walls for the generation algorithm. The first thing it does is properly initialize the MazeWalls and MazeCoordinates arrays. This is because to allow the user to enter odd numbered widths and heights, the parameters entered are the widths and heights in cells. To convert this to the actual array dimensions, we put both into the same formula: 2n+1, where n is the cell dimension. The + 1 represents the first wall. The 2n then represents that for every cell requested, we add a passage and a wall.

The arrays are then looped through. Open passages should only be on coordinates where both the X and Y position are odd, so the if statement uses the mod operator to check this. All other cells are initialized as walls. A Coordinate is also created for each cell and added to the MazeCoordinates array. In this way, the waffle shape required for generation is created.

### Recursive Backtrack Algorithm

See [PAGE] for this algorithm.

The recursive backtrack algorithm uses the call stack as a method of backtracking through the maze: the top coordinate on the call stack is the current position, and if there is no unvisited neighbour cell to move to, the top function call on the stack finishes, so it is essentially being popped off the stack and the current coordinate is the previous cell.

The algorithm first sets the current cell to be visited. It then initialises a list of all unvisited neighbour cells via another function. The GetUnvisitedNeighbours function checks for each of north, south, east, and west, first if accessing the cell would throw an index out of bounds error, and second, if it is visited. If it is accessible and unvisited, it is added to the list. If the list is empty, the constructor has reached a dead end, so the function does not enter the loop and the subroutine ends here. If there are cells, the while loop begins. If there are cells that are not visited, the constructor picks a random cell from that list of cells. It then destroys the wall between it and the target cell by averaging their X and Y coordinates. It removes the target cell from the list and calls the BuildMaze function on the target cell. In this way, every cell in the maze is visited before the call stack is empty.

### Growing Tree Algorithm

See [PAGE] for this algorithm.

This algorithm is unique since it does not make use of the Visited flag. It instead manages 2 lists of coordinates to decide where to go.

Firstly, the 2 lists are initialized and a random cell in the maze is selected to be the starting point. It is added to the list of “active cells”. An active cell is one which is on the border between visited and unvisited cells. Now the algorithm has 1 active cell, it will not end until there are no active cells, which will only happen when there are no unvisited cells left in the maze. The constructor cell is randomly picked from the list of active cells. Then, a list of that cells unvisited neighbours is fetched: here, this is a list of neighbouring cells not in the list of active or visited cells. If there are neighbours, a random one is selected as target. The wall between the constructor and target is broken by removing the wall at the average of their x and y positions. The target cell is now an active cell, so it is added to the list. If there are no neighbours, the cell must no longer be active, so it is removed from the active cell list and added to the visited list. Now, the process repeats, picking a new active cell and moving to its random neighbour, until all active cells have been exhausted.

### Wilson’s Algorithm

See [PAGE] for this algorithm.

This algorithm uses randomized loop-erased walks (henceforth RLEW) to produce a uniform spanning grid – the maze will be completely unbiased in the length and number of its corridors.

To account for the fact that the constructor breaks no walls until the end of the RLEW, the definition for unvisited in this algorithm is a cell with all 4 of its walls intact. The Visited property is still used, but it now marks cells that are currently a part of the maze and assists in removing loops during the RLEW.

During InitMaze, as well as generating the waffle, a list of all coordinates in the maze is initialized.

The first RLEW is slightly different to the rest, as it always walks from the start Coordinate to the end Coordinate. A list of Coordinates, Path, is initialised and the constructor, starting at the start Coordinate, is added to the list. Then, while the constructor is not at the end Coordinate, it randomly moves around the maze, adding each cell it moves into to the Path and marking it as Visited. If it moves into a Visited cell, it has looped, so it begins backtracking, unVisiting all cells it backtracks through until it reaches the Visited target cell. This erases the loop. When the path eventually reaches the end Coordinate, the Path list is iterated through, destroying all walls in the path. Then, each coordinate now in the maze is removed from the list of all cells in the maze. This is because this list will be used to pick random starting points for subsequent RLEWs.

After the initial RLEW, RLEWs are completed until there are no cells left in the cellsInMaze list. These RLEWs are different to the initial one, as they do not have a specific endpoint. They randomly walk until they reach a Visited coordinate, which must be a point in the maze. The path of this walk is then added to the maze in the same way as the initial RLEW.

## Maze Solving

### Depth-First Search

Depth first solve uses a stack of Coordinates to navigate through the maze. The algorithm is very fast but does not generate the best solution in a labyrinth maze, since it’s path can be summed up as ‘hugging the left wall’.

The solver is placed at the start Coordinate and is added to the solution stack. Its Coordinate is set to be visited. Then, it gets a list of its unvisited Neighbours by checking the Visited flag of its surrounding coordinates and if they are passages. If there are unvisited passages in the list, take the first one by moving the solver to it, adding the new position to the stack, and setting Visited to true. If there are no possible directions, the solver has reached a dead end, so it backtracks by popping the top element of the stack until there are unvisited neighbour cells. This process is repeated until the solver has reached the end Coordinate.

### Maze-Routing Algorithm

See [PAGE] for my implementation.

My implementation of this algorithm is a heuristic algorithm that finds a good solution to a labyrinth maze. It uses a stack and the GetManhattanDistance function for Coordinates to decide which path will be the best to take.

The algorithm starts with a Coordinate solver and a solution stack. The solver is added to the solution stack and the cell is marked as visited. Then, while the solver is not at the exit, the GetUnvisitedNeighbours function is called. Here, it returns a tuple of the Coordinates of unvisited neighbours and a char representing the cardinal direction they are from the solver. If there are no unvisited neighbour cells, the solver backtracks through the maze until it lands on a cell with unvisited neighbours. If there is only one path, the solver moves to it, adds it to the solution, and marks it as visited. If there are multiple paths, the TryPaths function is called and passed the list of potential neighbour cells. In this function, each potential neighbour’s path is tried in sequence until another fork in the path is reached. At each fork, the Manhattan distance is found between there and the exit, and the bestPath tuple is updated if the distance is lower than the best path currently found. Then, once each path has been tried, the path direction of the most productive path found is returned. The main solver then moves down this path. This whole process is repeated until the maze is solved.

## Logging in and Registering

See [PAGE] for the implementation of these features.

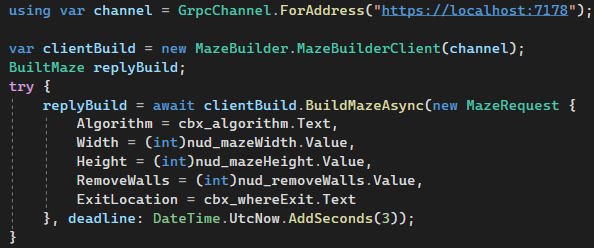
When the user presses the Register button on the registration form, their credentials are not immediately sent to the server. First, their credentials are checked:

* The server is queried as to whether the username is taken.
* The password length is checked.
* The password is checked for special characters using Regex. It must contain at least 1
* The password is checked if it matched the Confirm Password box.

The credentials are processed only if these checks are passed. The first step to processing the credentials is hashing the password. We call the HashPassword subroutine [[1]](#footnote-1)to do this. The salt is randomly generated with a length defined by the keySize global constant. The hash is then produced by the PBKDF2 algorithm with a number of iterations defined by the iterations global constant. This hash is then passed to the server as a hexadecimal string, as well as the hexadecimal conversion of the salt. The hex conversion is necessary as the hash and salt are both byte arrays, which are not supported types in the protocol files. The server creates a new record for the new user and stores their username, password, and salt under a new userID.

When the user attempts a login, the username and password entered are sent to the server’s LoginHandler service. The server uses SQL to find the stored password and salt hashes associated with the entered username. It then hashes the password it received using the same salt and algorithm parameters. It compares the hashes using the FixedTimeEquals method, so the state of the server cannot be guessed via the length of time the comparison takes. If the hashes are the same, the user is granted access and the clientside globals username and userID are updated. If the credentials are incorrect, the message “Username or Password incorrect!” is displayed. The message is ambiguous so the user cannot guess usernames and passwords and get information from it.

## Client-Server Interactions



Here is an example of a client call to the server. There are 3 steps to making a server call, and these are the same regardless of which service the client is requesting. Firstly, a communication channel is established with the server, using the address specified in the server’s launchSettings.Json file. The using keyword is used so the channel is properly disposed after use, to avoid memory leaks. Next, using the channel, the service to request is specified. Finally, the client calls the function of the service, passing in the request object with the relevant parameters. The await keyword is used here, and the method encompassing this code is an async method. This is because the time the server will take to receive, process, and transfer the data is uncertain, and we don’t want the client interface to hang while it is waiting. The async and await keywords allow the user interface to be interactive whilst the server manages data.

## Other Algorithms

These other algorithms are to do with managing the form interface.

### Connectivity Status

See [PAGE] for this algorithm.

This algorithm updates the label that notifies the user whether they are connected to the server. It has a polling rate of 10 seconds to not overload the server with requests. It uses multithreading to not lock up the main form, since it calls a while true loop. To do multithreading, an async lambda function containing the code we want to be ran is passed to the work item queue of the thread pool. In this code is a network call that tries to ping the server. If it manages to ping the server, another lambda is invoked which updates the label to tell the user they are connected. It also updates the Request Maze button so the user can send mazes. If the server fails to respond by the deadline, the HandleServerError method is called. A second catch block is required due to the unpredictable nature of the ObjectDisposedException (See “Multithreading Exceptions” for more detail). The loop then sleeps for 10 seconds before repeating.

### Setting the Size of the MazeDisplay Form

The private function SetDisplaySize is called on the load event of the MazeDisplay form. This function handles the dynamic sizing of the said form to ensure all the buttons are visible regardless of maze size.

The width of the form is set first. This uses a ternary operator to decide whether to size the form based on the maze size or the button panel size. It calculates the pixel width of the maze plus the pixel size of the margin between the table layout panel and form and checks if this size is greater than the size of the button panel. If it is, the width of the form is set to the size of the maze and margin. Otherwise, it is set to the size of the button panel.

Next, the height of the form is set to the pixel height of the button panel plus the maze height and margin. The panel containing the maze is then sized appropriately, and placed at a point so the left, bottom and top margins are correctly sized.

1. Sourced from https://code-maze.com/csharp-hashing-salting-passwords-best-practices/ [↑](#footnote-ref-1)