Guide Agent

Implementation Decisions

When implementing the logic component of the Guide Agent (guideagent.c) we decided it was best to factor a lot of the code out of the main game loop to abstract these functionalities from the gameflow itself. For this reason, we separated the logic into six main functions with ~fifteen helper functions. The 6 main functions are:

1. main
2. game
3. handleMessage
4. handleHint
5. sendGA_STATUS
6. sendGA_HINT

And each has its own role in the gameflow. Along with handleMessage we implemented a function dispatch table, called “opCodes”, to call a specific handler based on the opCode in the parsed message. handleMessage really only handles the data, using parseMessage to parse it into a message_t struct and the function dispatch table to actually handle the message itself. The handler functions almost all have their own validation function as well, based on the requirement spec, that simply check that all necessary fields have been initialized before the handling function attempts to use its data. If these functions return null, the guide agent simply ignores the message that was sent to it.

In terms of data structures, the Guide Agent does not really use any of its own. Most of its data is stored in its own team’s team_t structure that implements sets to hold various important types of data for the display (clues, field agents, etc.)

Limitations

As stated in the README, the Guide Agent has a few limitations of which the user should be aware.

1. Implementing the graphical interface using ncurses causes still reachable memory at program exit.
2. Implementing the graphical interface using ncurses prohibits the programmer from using stdout for anything except the interface itself. For example, if a bad message is received and ignored, the user will not know that it received a bad message because there it nowhere to notify them.
3. Because of our implementation of how to handle a GS_CLAIMED message and how clues are displayed in their window, sometimes the clues do not completely disappear when their respective krag is claimed (usually depends on length of old clue vs. new clue).
4. Because of the usage of select() to read from stdin and the socket and how we implemented how often to send a GA_STATUS to the server, this behavior can be

sporadic at times. It depends on the activity and frequency of the two inputs, as well as the timeout parameter given to select. This usually varies based on how often the Guide Agent sends hints and how many Field Agents are on their team.

Guide Agent – display.c

For the GUI component of the guide agent, we decided to place all the GUI functions into one single module called display. The display module is structured around five main windows: the map window, the input window, the clues window, the revealed string window, and the game stats window. All methods in the display module are designed to work for one particular window.

Assumptions

- There exists a map file called campusmap in ASCII code of size 90x45 in the guide-agent directory
- The window terminal has to meet the following minimum size: 126 width and 52 height. If the terminal window does not meet the following minimum size, then the GUI wont function and display properly
- Assume that hint is no more than 140 characters

Limitations

- Because there are only seven colors in ncurses, so the colors for the field agent players will repeated if there are more than seven field agents
- Only works with terminal window of size 126 width and 52 height
- In the user input window of the GUI, backspace is not supported
- The GUI only works with the map provided by the assignment

Field Agent

Implementation:

When implementing the Field Agent, we decided to write the following modules:

- main
- my_dialog_window
- p_message
- key_assembly

Each of these modules provides different vital aspects for the gameplay. The key_assemy (provided) includes the enum with the AppMessage keys, which are used to tell the smartphone proxy environment what kind of messages are being sent, and whether or not to send them over to the Game Server. Next, the my_dialog_window module has two (non-static) functions: my_dialog_window_push(char *text) and my_dialog_window_pop(). These push and pop popup dialog windows that contain text information for the user to see. They are used by many aspects of the game, such as tell the user when they have successfully claimed krags, have connectivity errors, or the game status. The p_message module is essentially the same as the

message module (see Common) that other parts of the game use, but instead of describing some values as ints, it treats every aspect of its respective p_message struct as strings. Only having to deal with strings makes coding the pebble much easier, as sscanf is not supported by pebble. Finally, the main module has the game logic and sets up the basic menu GUI outline. It contains the main function and handlers for different events that could occur during the game, such as different messages from the server, or button presses from the user.

Limitations:

1. The Game Server and proxy are must be up and running before the “Join Game” option is selected from the initial menu.
2. It is possible to “join” a game when none are running (as in reach the in-game screen), but pressing the select button, although selecting any option on the in-game menu will result in going back to the initial pre-game menu.
3. Hints may not have the ‘~’ character in them.
4. The game over dialog window does not always push when the end of game sequence is triggered (from receiving a GAME_OVER message).

Game Server

While it seems like a lot of code, the game server is structured and implemented in a very simple way. A main loop receives messages through the game server’s socket and handles each message based on its op code. The game server also makes use of the team, krag, and message modules to simplify and standardize how it deals with data across its many functions. Almost every function in the game server is a message handler, a message or player validation function, or a function that sends a specific type of message. In this way, the game server is can be simplified to three points of pseudo code: 1) receive message 2) validate message 3) send response.

Limitations

1. It is impossible for a Field Agent to win the game without a Guide Agent on their team.
2. The game cannot end until a team wins.
3. If the Guide Agent quits the game and tries to rejoin with gameId 0, the server will treat this as invalid because the Guide Agent is already part of the game.

Common Modules

The common modules were implemented in order to abstract unnecessary code away from the functionality of the individual agents, while simultaneously giving common code for all major components of the game to share if they need it. For example, the network module is ubiquitous

in this project, being used to connect to the server/specific agents and send messages back and forth.

Message

The message module was implemented to allow for easier and simpler access to the various fields of any given message. This module almost completely abstracts away the handling of semantically incorrect messages, storing an errorCode != 0 in the message struct if any area of the message is incorrect. The agent/server can then handle the message as it sees fit based on these error codes. The code is long and tedious, but error checks heavily and returns a clean struct to the caller.

Log

The log module is even hard to call a module. It is a function we decided to abstract away from the normal handling of messages to make it easier to log messages with just the message, the connection of interest, and the file path to the log file. The log file's existence is checked before any message even has a chance of being logged, so there are little-to-no errors that could occur in this functionality of the game.

Word

This module was taken from a team member's TSE project, used to normalize the names going into sets so they are more easily accessible and less need be assumed about each key into a specific set (namely, the Field Agent set in a team_t struct).

Network

The network module allows messages to be sent and received by the main game components. It includes the code that starts the server, opens the guide agent's socket, sends messages via the UDP socket, and receives messages via the UDP socket.

Team

The team module contains all the logic for creating teams, adding guide agents and field agents, and related functions such as adding krags or updating location. The team module is coded in an object-oriented style with objects being the team, guide agent, and field agents structs and with getter functions to access the information stored in each struct. The entire team module revolves around one data structure: a hashtable. In this hashtable, the key will be the team's name and the item is the corresponding team struct, which in turns stores the guide agent, field agents, and every other aspect of a team. Even though there are only a few teams for testing

purposes, we decided to use a hashtable in case the clients wanted to scale up the size of the game and having many teams participating in the game.

Assumptions

- assumed that 50 is a good size for the hashtable of teams.

Krag

The krag module contains all the logic for saving krags to a hashtable, getting the secret string, revealing characters, and providing clues to krags. Just like the team module, the krag module is coded in a object-oriented style with objects being the krag struct. The entire team module revolves around one data structure: a hashtable. In this hashtable, the key will be the kragID and the item is the corresponding krag struct, which in turns store some information about krags such as their location or clue.

Assumptions

- assume that 20 is a good size for the hashtable of krags
- assume there is less or equal than 20 krags
- assume the kragfile format follows the format of the requirements specs
- assume the secretfile is at most 140 characters
- the length of the secret string (n) may be assumed to be greater than the number of krags (k), as needed for the reveal
- assume that the randomClue method wont be called for a team who has found all krags because the method will get stuck in an infinite loop.

Limitations

- works only with less or equal than 20 krags.