Anti-Battleship

Kevin Hsiue, Colin Taylor, Zach Hynes

May 10, 2011

1 Reflections

Group Reflections

Our group found this project to be incredibly rewarding and satisfying. The challenges at the outset of the project derived largely from our inexperience with Swing and networking; we had difficulty envisioning a system that could support two users communicating over a network, let alone multiple users independently initiating multiple different games with each other.

We initially tried to develop a client-server model, in which all users playing games over the system connected to a central server. However, we found that this setup produced extra complications; the server would have to maintain information about which clients were playing against other clients at any given time in order to route messages properly from one user to another. As we gained experience with networking, we found it much easier to adapt our code to expectations. For example, when we realized that the class listening for incoming connections had to exist independently of the class initiating outgoing connections to comply with the tournament regulations, we were able to split the original class, MasterServer, into an ABIncomingGameServer and an ABOutcomingGameServer to comply with the tournament standards.

We were particularly satisfied with our usability strategy; by offering early, functioning versions of our game to approximately ten of our peers, we received valuable feedback that enabled us to improve upon our GUI to make it more user-friendly. While we would have liked to develop more rigorous testing protocols for certain aspects of our game, the built-in checks that we developed on message sending through the development of our protocol classes and protocol-state maps made the game more robust, as did the locking mechanisms in place within the GUI.

We found that the size of our code grew quickly as we dived into development. Ideally, we would have cleaned the code and written specifications at more regular intervals. However, we found that having to explain the code to other members of the team every day or every other day forced us to revisit and rethink our implementation. The collaborative process of code development greatly enhanced our final product. We developed sharper sensibilities for when it was appropriate to modularize certain aspects of our code and when it was appropriate to maintain larger classes to facilitate easier access to relevant information.

Our design benefited tremendously from a visually appealing, uncluttered GUI that was improved by our experimentation in usability. Consistent testing throughout the project of different components (the network architecture and the GUI) made for near seamless integration of these two portions of the project in the final week. We are proud of our Antibattleship game, worked well as a team, and met our collective goals.

2 Usability

1. Initial Panel

   The Initial Panel was tested by making sure that only legal parameters allow progress to the next stage, and ultimately the send parameters capability. Otherwise, illegal parameters will lock the next step and disallow the user from breaking the program. After using actual testers to try out the program, instructions were added that were updated after each entry, which verified or rejected the parameter and provided further instruction. This seemed to fix the confusion in the screen and ultimately proved to be sufficient for this panel. Testers after this addition had no problem understanding and setting up the game board. Extreme test cases were used, such as 1x2 or 2x1, which all passed without problem.

2. Set Up

   The Set Up screen was also tested thoroughly by real testers. The biggest problem with this panel was that it was not immediately clear as to how to place ships and what legal placements are. Furthermore, there were initially some GUI issues with ship placement with edge cases, such as ships of size 1, but these problems were resolved after being pointed out by further testing. However, the issue with usability was finally resolved by locking the process of placing ships similar to the Initial Panel as well as providing concise and clear instructions in the panel as well.

3. In Game

   The In Game panel did not require that much testing besides just to make sure all the components worked. Ultimately, the board was easy to test and debug since all the individual components were created and put together after it was assured that they were debugged. The graphical representation sometimes proved challenging using Swing, but was ultimately completed with an optimal balance of aesthetics and functionality.

3 Evolution of the GUI

1. `ConnectionFrame` The Connection Frame is the first panel a user sees upon executing the program. Initially, the frame itself only consisted of a two text fields and an initiate game button. Later on, the frame was decorated with the title banner, a graphic, and streamlining of the text fields and game initiation buttons. Furthermore, an AI option

was added as well. The two text fields for host and port are by default filled to connect with the reference player, but can be changed by the user. As far as debugging goes, the Connection Frame was rather straight forward and proved to be simple for users so no drastic changes were made to the functionality.

2. `Initial Panel` The Initial Panel is the second panel that is brought up after the successful initiation of a game. The frame itself has maintained the same structure since its creation, but has evolved mainly due to testing responses. The most significant change that contributed to functionality was the verification/locking/unlocking process that was implemented to guarantee legal parameters were entered. The goal was to create simple step by step instructions, as well as responses, for the user to create a valid game. A user is forced to first enter a valid board size, and then set the board, which locks the board entries. Then, the user is forced to enter ships sizes, while both maximum number and sizes based on the board size are readily visible on the screen. Once a single ship is entered, the send game parameters button is enabled and should the user desire, a game can be started with that single ship. The user will be prevented by the GUI to enter illegal ship sizes as well as entering too many ships. Also, a clear all button is always present to clear all the inputs entered so far, which resets the entire procedure. With these locks in place, it is impossible to send illegal parameters. Furthermore, the instructions and responses were proven in test cases with actual users to be extremely helpful and straightforward.

3. `SetUp` The Set Up screen is the third panel to be displayed after a successful transmission of legal parameters from the Initial Panel. In this screen, the user is asked to place his ships on the board. A user selects a ship size on in the list, and then clicks on the board to view valid, green square placements. This process is intuitive, and allows for real time checking of valid locations of the ship. The instructions are well-written and immediately seen, detailing each step in placing a ship. This was added due to a lot of confusion from testers on how and where to place ships. With these text instructions, it is very clear and easy to follow them and learn how to place ships. Furthermore, the GUI is triggered so that there is no ship placement without selected a ship, and after each ship is placed it is cleared from the list. Furthermore, should a new ship be selected, the board orientations are cleared, while the ships remain. This panel proved to be one of the more difficult panels to debug and make user friendly. After several revisions, it has reached a very usable state where any individual can simply follow the on screen instructions. Finally, similar to Initial Panel, a clear all button is present to clear all the ships placed so far and also repopulate the Ship List.

4. `InGame` The In Game Panel is the actual game representation. Though it has gone through several aesthetic changes, the overall layout has not been severely edited. However, the additional extensions of the game history as well as the capability of chat resulted in some renovation of the overall GUI. The game functions after clicking on the opponent board GUI and clicking on fire, which sends the message. After a small number of testers stated that it was initially confusing to start the game,

instructions were added to the history box in the very beginning, and would disappear as the game actions carried on. Though not planned, this worked out very well, since those instructions are not necessary throughout the entire match and therefore will be replaced by the game history. Overall, the game GUI is pretty intuitive and follows a very straight forward progression.

4 Architecture: Implementation

Our peer-to-peer network architecture runs from the `BasicMain` class. The key features and classes for the control of gameplay are organized in Figure 2.

1. `ClientManager` (holds `ClientManager.sendMessageQueue`,`ClientManager.messageProcessor`, `ClientManager.protocol`)

   ClientManager is the thread that manages all user interactions with the game. ClientManager is spawned when a game is initiated with a foreign opponent or an opponent connects with the ClientManager. Incoming and outgoing messages are placed on the sendMessageQueue (ArrayBlockingQueue<Message>) of ClientManager, which handles each message using a messageProcessor (MessageProcessor).

   The MessageProcessor class uses current state information to determine how to handle incoming messages based on the current state of the game. In MessageProcessor, the board models and board GUI's are updated to reflect new information, and automated messages are sent if appropriate. For example, MessageProcessor could add messages to the queue like:

   - `accept-victory` or `reject-victory`: the anti-cheating protocol does not (or does) detect cheating
   - `results fire`: the opponent has hit one of the host ships without sinking the ship

   MessageProcessor will also change the current content pane of the ClientManager JFrame, gameFrame, as appropriate.

   SendReceiveProtocol instances maintain a map of states to allowed messages; this helps ensure that messages being sent at the wrong time in the game do not affect the game state and instead will be handled as syntax-errors.

2. `ABIncomingGameServer` and `ABOutgoingGameServer` (classes implementing the separate listening and initiating functions of MasterServer)

   Instances of these classes are run in the `BasicMain` method to listen for incoming connections and initiate outgoing connections, respectively. They can both spawn instances of `ClientManager`.

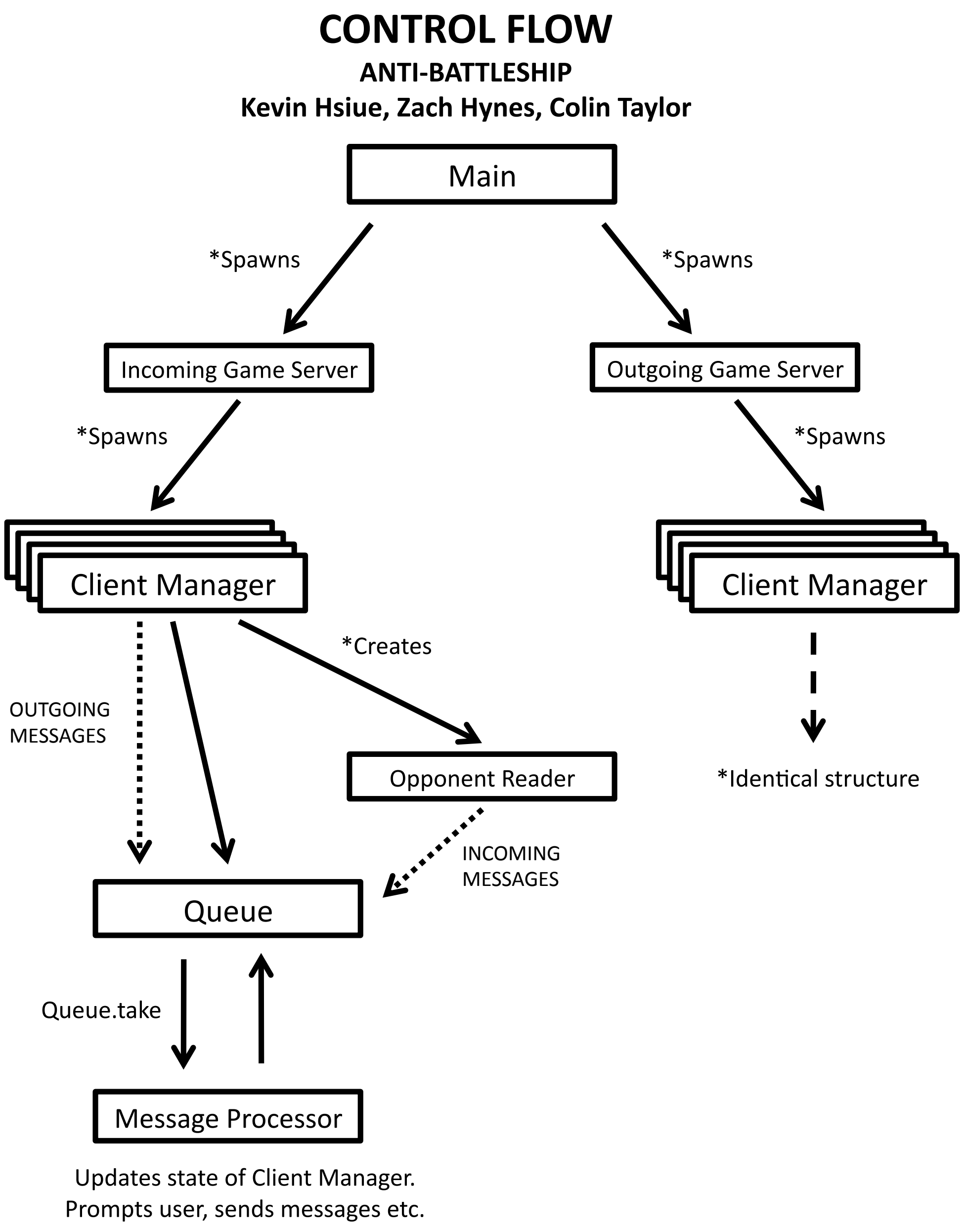

Figure 1: Control Flow of Gameplay over the Network

5 Testing

One of our goals was to be as complete as possible in testing. Throughout the project, we used various testing methods. By the end of the project, however, we spent quite a bit of time making sure our testing was as thorough as possible. We implemented modular testing and complete system, end-to-end testing as much as possible. We felt it prudent to outline a few of our methods.

5.1 Message syntax

One of the areas we focused early on was the testing of our messages. We made sure that our code will always send message with correct syntax. It should also parse messages correctly, and handle errors in receiving messages. We tested cases to make sure we could only send messages at appropriate times- for example, that we could only send target messages when it was our turn. We also made sure that we only send messages by passing it through our message generator functions. We also extensively tested different incoming messages. While we are in certain states, we can only accept certain types of messages. We also made sure that any type of syntax errors, including those that through exceptions during parsing etc, returned a syntax error. Other errors are rejected, and game errors are thrown in certain cases, such as targeting the same square etc. We built automatic test case for message syntax. We extended it to include our extension (chat) and salvo mode.

5.2 Handling victory

Another area we did thorough modular testing on is on handling victory. We went through a variety of ways that the opponent could cheat, and made sure that we caught these. In particular, we tested the following ways that an opponent could cheat: Incorrect hashing (initial hash and final hash were inconsistent) Inconsistent with their reported internal state (they told us a miss when it should have been a hit, etc) Not having a valid board (ship) placement (they violated the rules of the game).

5.3 GUI and Overall System

: As discussed in our usability section, we put a lot of emphasis on testing and improving the GUI. We felt like this was very important because this is the eventual end-to-end test of our entire system. To test it, we came up with a list of edge cases and played a variety of games to see how our GUI performed. For example, we tried starting games with different parameters, clicking around, closing it midgame, starting multiple games etc.

Examples of scenarios we tried were:

- Small board, dense/large ships
- Large board, dense ships

- Large board, few ships
- Small board, few ships
- The same as above, repeated for salvo.
- Same as above, repeated for AI.
- Repeated as a 2nd connection.
- Repeated with reference player.

In addition, we had friends of ours not familiar with the system try playing different games and instructed them to try different things. We fixed problems we saw, and documented our fixes.

5.4 Overall System w/o GUI interaction

: We originally felt that the best way to test our system was to write test cases with hard coded drivers. However, we realized that it was very hard to write test cases to check the communication for entire games etc. Instead, we chose to play AI vs AI games, and display the games on the GUI. It was easy to test using our game history box, and we could see the AIs shots. Doing this also enabled us to quickly set up games in the same way that a user would. As we developed our game, we utilized command-line and text-based ship setup in order to test our protocol; this was useful to the simultaneous development of all components of our project.

5.5 Testing Results

All of our tests worked to our satisfaction. All of our automatic test cases (such as Junit Tests) passed as expected. All of our GUI testing allowed us to find and fix small errors. Our AI player works to our satisfaction.

The one thing we werent able to fix in testing was a bug in how our program acts when we close JFrames. We found out that if we close the game window, the socket will not close, and problems can happen if you try to start another game from the SAME (not spawned!) master window. We tried to fix it, but we ran into problems trying to close the Opponent reader thread- this caused our program to freeze when we tried to close the socket to resuse. Unfortunately, we ran into the deadline before we were able to fix this, as we found the problem during later testing.