Network Architecture and Abstract Designs

Zach Hynes, Colin Taylor, Kevin Hsiue

April 28, 2011

1 Network Architecture

Our implementation of antibattleship utilizes a peer-to-peer (P2P) network architecture. All players run instances of `MasterServer`. The `main` method of `MasterServer` listens for incoming connections and allows the host to initiate network connections with other players. Once a connection is established between two players over the network, the `MasterServer` spawns an instance of `ClientManager`. When the player wishes to begin the game, a new thread is initialized to run the `ClientManager`.

Messages are transmitted directly between players. Once created, the `ClientManager` spawns an instance of `OpponentReader` and starts the thread. `OpponentReader` reads all incoming messages from the other client using a `BufferedReader`. The message will be parsed and handled; the `ClientManager` will be notified or updated in accordance with the message type. For example, if the opponent says that one of his or her ships has been hit, the message will be routed to the `BufferedReader` in `OpponentReader`, read out, and the BoardModel (a member of the ClientManager) will be updated accordingly. A separate thread for reading messages is vital to ensure that players can be active (ie, selecting moves, writing chat messages, etc.) while awaiting a response on a message.

2 Network Protocol

We will use the standard network protocol to implement the basic game functions. Our group has also chosen to implement chat functionality. Using a textbox on the gamescreen, users can type and send chat messages through the same socket over which game commands are sent. Each text message sent in the form `<chat> some message </chat>`. Opponents must employ the same network protocol in order to engage in chat conversations with the user. We are considering how to implement a multiuser chat room with the client-server model; given that our current architecture is based on the peer-to-peer framework, this could prove difficult.

If the users agree to play a 'speed' battleship game, they must transmit information about the time spent on each move to the opponent over the same network connection. Timing information is sent over the network in the form `<time> (number of seconds) </time>`.

Each user can thus keep track of the time spent on moves by both players and declare victory when the opponent's accumulated time goes over the specified time limitation. One can imagine implementing restrictions on the maximum time per move in a similar fashion. It should be noted that while users could time opponents on their own clocks, this would lead to conflicting times recorded by the players due to the time associated with sending moves over the network. Our timing mechanism will experience none of these problems.

3 Design Choices

We explored several alternative models and tweaks before concluding on our model for network connections. We actually spent a considerable amount of time discussing different models, and only selected our protocol after running into barriers with our other designs.

The first alternative design we ended up rejecting was using the server to always pass messages. If client A and B want to initiate a game, we considered having both talk to the server, and have the server rely messages. This would require a more complicated system for A to tell the server which client the message is intended for, and the ref_protocol does not have that information standard. We could append this if a certain mode was enabled (like we are thinking of doing in our speed-antibattleship mode, for example), but for simplicity in compatibility with the standard protocol we chose to make everything peer to peer.

We also ended up discussing several models for the necessary classes and threads needed to make our p2p model work. We chose to implement a Master Server, and a Client Manager (instances spawned upon connection. We had discussed having a separate game class that spawned upon an accepted game, but chose to combine that with a instance of CM for simplification.

We are using a buffered reader, `OpponentReader`, to read and handle messages as they come. We considered breaking up incoming message handling between a queue handler thread and the reader thread to allow for asynchronous handling of messages and for messages to be received while the handler is processing a previous message. However, the fact that the class reading input over the network is a `BufferedReader` renders this more convoluted setup unnecessary. We chose to allow the `ClientManager` and the `OpponentReader` thread to both access the model, in order for GUI events and message events to be handled concurrently. We discussed whether this would cause problems. We will implement tight controls within `ClientManager` to ensure that no improper messages (out-of-turn, etc.) can be sent.

4 AbstractDesigns

This section gives an overview of major classes utilized in our implementation.

- `MasterServer`
  - Opens up a port
  - The `main` thread allows for users to establish network connections with each other

  - Spawns `ClientManager` threads to run individual games

- `ClientManager implements Runnable`
  - The `ClientManager` encompasses all user interactions with the game
  - Possesses `BoardModel` and `BoardGUI` members that represent the user's interaction with and knowledge of the game
  - Uses an output writer, `PrintWriter`, to send messages to the opponent
  - Maintains a basic integer variable representing the state of the game (initiation, ship layout, game play) that determines how the user interacts with the opponent

- `OpponentReader`
  - Spawned from `ClientManager`, an instance of `OpponentReader` reads and handles all incoming connections
  - Uses the socket already established in `ClientManager`

- Model: BoardModel
  - The BoardModel represents the data structure that will store the information necessary for the game to function. Fundamentally, this will be achieved with a two-dimensional matrix .
  - BoardModel will accept commands, methods defined within itself, to alter the state of the board. This will be in the form of a legal input of the BoardModel which will then alter the state of the specified position.
  - Should generate appropriate responses to actions performed on positions with a certain state.
  - Keeps track of positions in different ArrayLists that represent each state; should the occupied (ships left) ArrayList be empty, then BoardModel should recognize that the game has ended.
  - BoardModel should accept inputs from the controller, and output the state of the game to the view

- View: BoardGUI
  - The BoardGUI is a visual representation of the model. Each element of the two-dimensional array will be represented as a panel of a certain color/graphic.
  - BoardGUI should register user inputs and decipher what/where the input occurs, and output that information to the controller.
  - Furthermore, BoardGUI will refer to the model to be constantly updated to the latest state of the game.

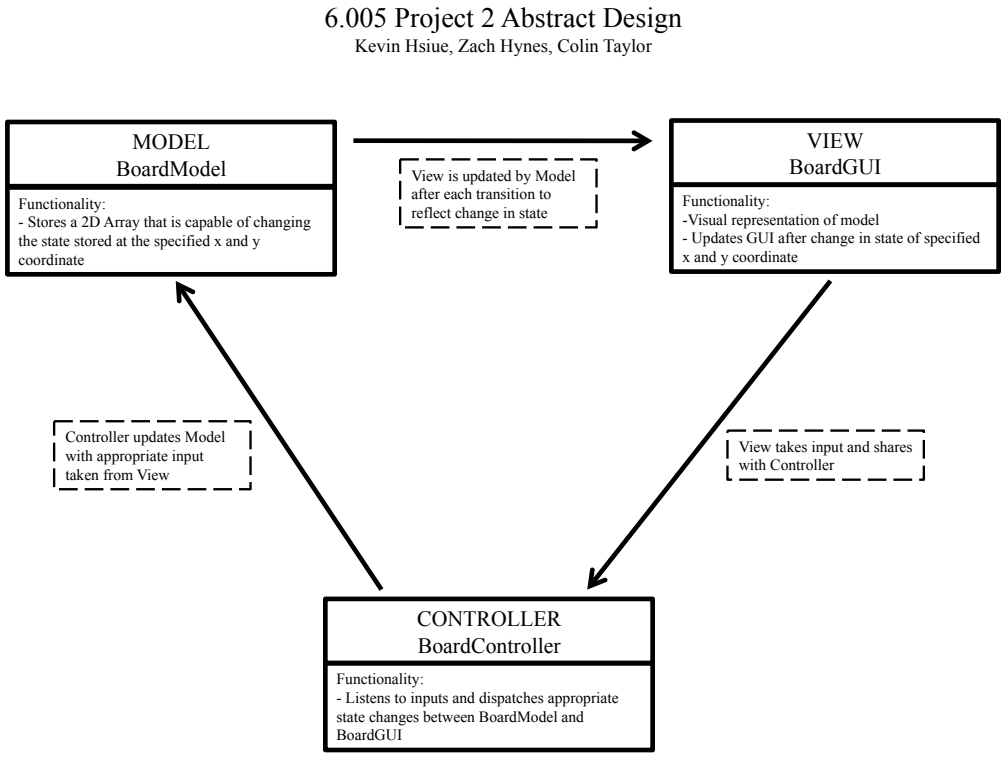

Figure 1: MVC Client Interface

- Controller: BoardController
  - The BoardController will handle (listen to) events that are created from user input to the GUI- clinking a square, button etc.
  - BoardController should listen to events subscribed in the view as inputs to a certain position, and communicate that information to the model.