Matthew Carrington-Fair
Matthew Epstein
David Stern

# Sharks and Minnows Final Report

## Primitive Plans (Original Idea)

Staying faithful to our original design choices, we created a fully playable distributed Sharks and Minnows terminal-based game in Python with the help of Pyro. In addition, we settled on using the original ASCII models for both the fish and sharks, and we continued the decision of organizing the client code into two distinct threads (one for fish control and writing to the board and the other for reading the board and displaying the board and current game status to the user).

## Current Components (Refined Design)

For the final version of our Sharks and Minnows game, we worked on optimizing the game's performance and adding aspects that make it appear more like a real game. At the most basic level, we changed the game's presentation so that the part of the screen comprising the board has a black background and the rest of the screen has a blue background. Below the board it reads Sharks and Minnows in a cool-looking ASCII font. To accomplish this formatting and all terminal output we used Python's curses module.

One of the biggest changes we made relates to how the game starts. Upon joining a game, the user is prompted for a username and is asked if he or she would like to start playing immediately or if they would like to wait for more players. By choosing the latter option, the user will be sent to a waiting lobby, represented as an empty board, where the user can "swim" around, waiting to be joined by other fish. Other users can join the waiting lobby until an entering user says that they would not like to wait for other players, at which point that user will enter onto the board and the game will immediately start. (Only an entering user can start the game. A user who has already entered the lobby can no longer choose to start the game. The lobby closely resembles the type of waiting room that we used for some of the puzzles.) A new player cannot enter a game that is already in session.

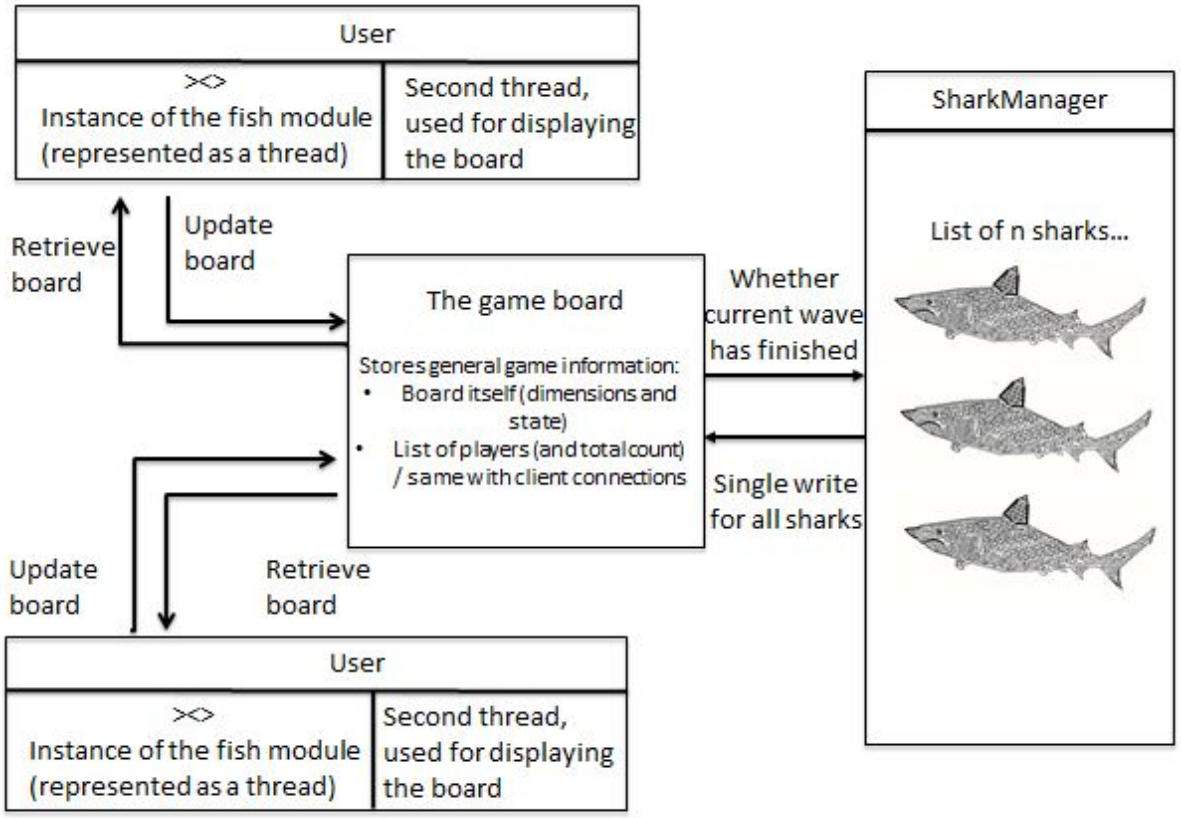
Once a game has commenced, sharks will be spawned in "waves" (as opposed to our previous design, when sharks were just randomly generated). Each wave features the same number of sharks as the wave number (i.e. the first wave features one shark, the second features two, etc.). For each wave, the sharks travel from the left side of the board to the right. This change gives the game more structure, so that players can more easily judge their success in each game. In the spirit of fairness, a fish's initial position on the board is no longer randomly generated: each fish will now always start at the same spot on the board.

Users die upon collision with a shark. So that users can differentiate their fish from other fishes, each user now appears on the screen as an ASCII fish with their username directly above the fish. It is important to note that for collision purposes, the username is not considered as part of the fish. In addition, so that the username does not limit a fish's mobility, the username will display only as many characters as possible. For example, if the length of the username is longer than that of the fish, moving to the board's right-most column will cut off the

characters in the username whose position in the username exceed the length of the fish. Moving back to the left will re-display the truncated characters. This is a better solution than the alternative option of preventing users with longer names from accessing the right-most column, since this gives users with shorter names an advantage. (Users cannot have names longer than ten characters though, nor can two users have the same name.)

Another piece of the game we modified was what happens when a player dies. Upon being eaten, the user's fish will disappear and a message will appear on the lower, left-hand corner, informing the user of their unfortunate fate. If the user is not the final player to die, he or she can still watch the other users play. Once the final user dies a timeout starts, where all users still watching the board can continue to watch for five seconds, at which point any remaining players will be exited from the game and the game server will be shut down. In addition, since users need to know the IP address of the server in order to join the game, the program now automatically prints the user's IP address upon starting a server.

The most important change we made pertains to the implementation of our sharks. Previously, each shark had its own thread, meaning that all the sharks and the fish were constantly trying to access the board. While this was okay for the first (and sometimes the second) wave, when there were more sharks on the board, all the threads competing for the board made the game too glitchy to be played. When the board became too crowded, sharks and fish were often rendered invisible. Our solution was something we call the "shark manager." Each shark still has its own position and speed attributes, but a single thread represents all of the sharks. This means that instead of each shark thread trying to access the board at the same time, all sharks are written to the board simultaneously, during a single write. This drastically improved the performance of our game.




Delivered Deliverables (Outcome)

We believe that we successfully achieved our minimum deliverable. Multiple users (representing different threads) can play the game, sharks successfully spawn on the screen, and collisions are handled well. While we did not achieve any part of our maximum deliverable, we did implement some aspects of the game that we did not originally specify as part of the minimum or maximum deliverable (eg. screen formatting, waiting lobby, username handling on the board). We also spent time trying to create a Heroku server (and then a Raspberry Pi) that could always host the game. This would have eliminated the need for a user to run the server and would have made it easier to connect to the board (since the server address would have been constant) but we were initially did not realize that to access the board object our nameserver for Pyro needed to be on the same network.

Design Decisions

Most of our notable design decisions—both the good ones and the bad ones—pertain to our implementation of sharks. In our original design, which we ultimately changed, the shark writing method was in the server file. When a shark needed to be written, a function was called which would change a single character on the board. This meant that each time we wanted to write a shark, we needed to individually update each character. The board is a Pyro object—and a call to a Pyro object is fairly expensive—so performance suffered. Our solution

was to create the shark outside of the server and then do a single function call, so writing the shark only needed to access the Pyro object once. What we consider our best design decision resulted from one of our worst ones: the shark manager (described in the Current State section) improved performance significantly and brought our game to the next level.

Division of Duties

We have all made an effort since the project was first designed to get together at least once a week to work on the project together. We all had a fairly good understanding of how the code worked, so if one person felt like making changes he could do so without jeopardizing the rest of the group's understanding.

An Irritating Issue (Most Difficult Bug)

Our most difficult bug (which we never resolved, so much as learned to work around) related to compatibility issues between Windows and Macs. For some reason, the Macs cannot connect to the game when the Windows computer is hosting the server, but the Windows computer can connect when one of the Macs is hosting. Towards the beginning of the project, we thought this was caused by some error in our code, but we have ultimately concluded that it is due to some issue beyond our control. Our "solution" was not testing our game with the Windows computer hosting the server. Sorry, Matt.

Finally, Our Files (Overview of files)

To run our program, the two most important files are game.py and client.py.

Game.py initializes the server for the program, which includes starting the Pyro nameserver and setting up our board object to be available to client programs. It does both by launching python subprocesses that can run in the background while the rest of the program can continue to the main server loop handling the game status and creating sharks. Upon successful start-up of both the nameserver and the game board, it'll output the IP that the nameserver is running on so client's can connect to it.

Board.py contains our Pyro class object. It is the centerpiece of our game, containing all information regarding the current game state (the board itself, whether the game has started, the number of players/their usernames, etc.) and with methods to update the game information and board through read & write operations. It is made accessible to both the server and client programs throughout the game.

To handle shark creation, the game server also has access to the shark model and shark class located in shark.py

Client.py handles the gameplay experience for the users. It requires the IP address of a nameserver, and upon start-up prompts the user for a username and whether they will want to

wait and play with a friend or play solo. From there the program branches into two threads, one to handle user input and controlling their fish object (obtained through fish.py), and the other to keep a clear display (with the help of the curses library).

Any other files include the ascii models for both the sharks and minnows, and our ascii art title string which we just decided to include as a python file containing a longstring for simplicity.