Clamball Final Report

Zack Amiton, Lucy Gramley, Mikayla Walsh, Calvin Will

12-17-24

1. Introduction & User Assumption
2. Revised Requirements
3. Revised Architecture
4. Sequence Diagrams (3)
5. Revised FSM
6. Revised Traceability Matrix
7. Testing Approach
8. LTL Requirements (2 Safety, 3 Liveness)
9. Environment Processes
10. Submission Files & Requirement Meeting
11. Unit Test Explanation
12. Reflection
13. Peer Feedback Appendix

1. A short introduction giving an overview of your project and what assumptions you are making about the user

For our project, we designed and assembled a playable version of Fascination, a parlor game from the 20th century. The game functions as a cross between Bingo & Skeeball—players compete to roll skee balls towards a 5x5 grid of holes (housed in a *cabinet*) and aim to have their ball land in specific holes such that they attain a 5-in-a-row (or, in Blackout mode, full coverage of the board). To recreate the competitive multiplayer experience of the original, we created two cabinets, allowing players to go head-to-head to race for the win. Each cabinet has an onboard Arduino, and uses Wi-Fi communication to send to and receive game updates from a central server.

In order for this to work, we are assuming that the user is able to roll the ball into the holes and is able to push a manual reset button after a hole is made. Additionally, we are assuming that only one ball is used at a time for each cabinet during the game. This allows us to have the game go into lockout mode while users are retrieving the ball. Lastly, we are assuming that the user is not trying to connect any other clients while the game is in progress. This means that the number of cabinets that are connected when the game start message is sent is the maximum number of cabinets that will ever connect during the lifetime of the game. Additionally, we assume that users are not malicious, thus only and always use the cabinet ID assigned to them by the server.

2. *Revised and fleshed out requirements from part 1 of the progress milestone report.

These requirements have been adjusted from the milestone report to reflect changes made to our FSM. We received full points and no comments on our requirements in our progress milestone report, so no feedback could be incorporated.

~

For the purposes of these requirements, ***cabinet*** is used to refer to the interface interacted with by an end-user, and is treated as client to a ***game manager***, which receives state updates from all connected clients, and makes updates as necessary. A ***game pattern*** defines the necessary goal state to be achieved by a cabinet, and is defined by a collection of holes to be ***covered.***

State changes will be *italicized,* while inter-device communication will be underlined. All messages are requests sent from cabinet to game manager, while replies are sent from manager to cabinet.

Game loop is used to refer to states during which a game is considered to be "in-progress." For a cabinet, this is the set of states: {*waiting_for_ball*, *ball_sensed*, *update_coverage*, *send_update*, *hole_lockout*, *send_heartbeat*}. Further, *end states* refers to the set of cabinet states: {*game_win*, *game_lose*}.

Finally, the exhaustive suite of messages that can be sent by a cabinet are detailed as:

- Handshake Initiation
- Game Start
- Hole Update
- Heartbeat

RO-A: On power on, a cabinet **shall** immediately be placed into *attempting state*, during which it **shall**:

a. Attempt to connect to the game manager by sending a handshake initiation message
b. Retry its handshake initiation message at regular intervals until an acknowledgement reply is received from the server
c. Be placed into *waiting_for_game state* after receiving acknowledgement reply containing a cabinet ID assigned by the server
d. The cabinet **shall** make all future requests using the replied cabinet ID

RO-B: On power on, a game manager **shall** immediately be placed into a *idle state,* during which:

a. On receiving a register message from a cabinet, the game manager **shall** send an acknowledgement message containing the ID assigned to the cabinet, and consider it a *connected cabinet*
b. On receiving a valid message from a connected cabinet, the game manager **shall** send a contextual reply based on the valid message type has been received
c. On receiving valid interface input from a user, the game manager shall update based on the message

RO-D: The game manager **shall** provide an interface to begin a game for all connected cabinets:

a. Post-beginning a game, the game manager **shall** respond to all game start messages received from any connected cabinets with an affirmative reply
b. Pre-beginning a game, the n orchestration state, the game manager **shall** respond to all game start messages received from any connected cabinets with a negative reply

RO-Q: The game manager **shall** provide a mechanism to change the *game pattern* used for by any **connected cabinets**

RO-E: A cabinet in *waiting_for_game state* **shall** attempt to initiate a game by sending game start request messages to the game manager at regular intervals

a. On receiving an affirmative reply from the game manager, a cabinet **shall** be placed into *initialize_game state*

RO-F: A cabinet in *waiting_for_ball*, on attaining coverage of a hole in its grid, **shall**:

a. Transmit a hole update message containing the metadata of the hole attained to the game manager
b. Update its display to indicate coverage of the hole
c. Enter a period of lockout until the cabinet is able to reliably attain coverage of another hole in its grid

RO-G: A cabinet in its *game loop*, on receiving a reply to a hole update message or heartbeat message, containing a winning cabinet ID (i.e. a non-null reply), **shall** transition into one of the *end states* based on the contents of the message

a. If the reply matches the cabinet ID of the current cabinet, the cabinet **shall** transition to *game_win* state and display a winning message
b. If the reply does not match the cabinet ID of the current cabinet, the cabinet **shall** transition to *game_lose* state and display a losing message
c. If the reply contains a null ID, the game should continue without transitioning to an *end state*

RO-H: A cabinet in an *end state* **shall** eventually return to *attempting* state, enabling future games to be initiated

RO-I: A cabinet in its *game loop* **shall** return to *attempting* state after a period of manager inactivity (non-receival of server messages) or user inactivity (non-receival of sensor input) no shorter than 5 seconds

RO-K: A cabinet **shall** be considered the winning cabinet if it achieves coverage of the given game pattern, as determined by the game manager

a. Valid game patterns **shall** include line and blackout, but can be expanded to include additional patterns
    i. Blackout **shall** require coverage of *all* holes on the game board
    ii. Line **shall** require coverage of at least one of:
        1. All holes of a given row index
        2. All holes of a given column index
        3. All holes having equal row and column index (i.e. the downward-sloping central diagonal)
        4. All holes with row + column index = 4 (i.e. the upward-sloping central diagonal)

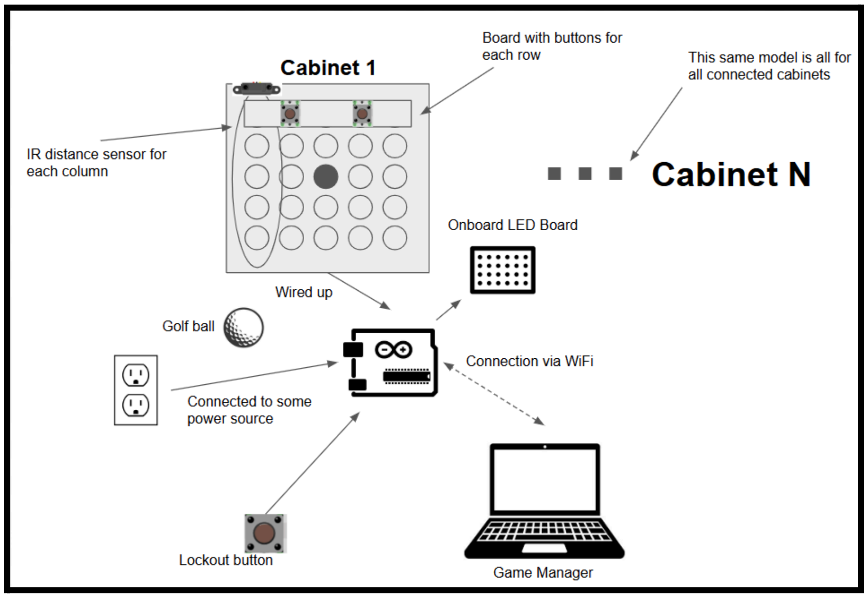
RO-L: A game manager with a started game, on receiving a hole update from a connected cabinet, **shall** determine if the received message places the transmitting cabinet into a winning configuration based on the active game ruleset, becoming the "winning cabinet" unless one already exists

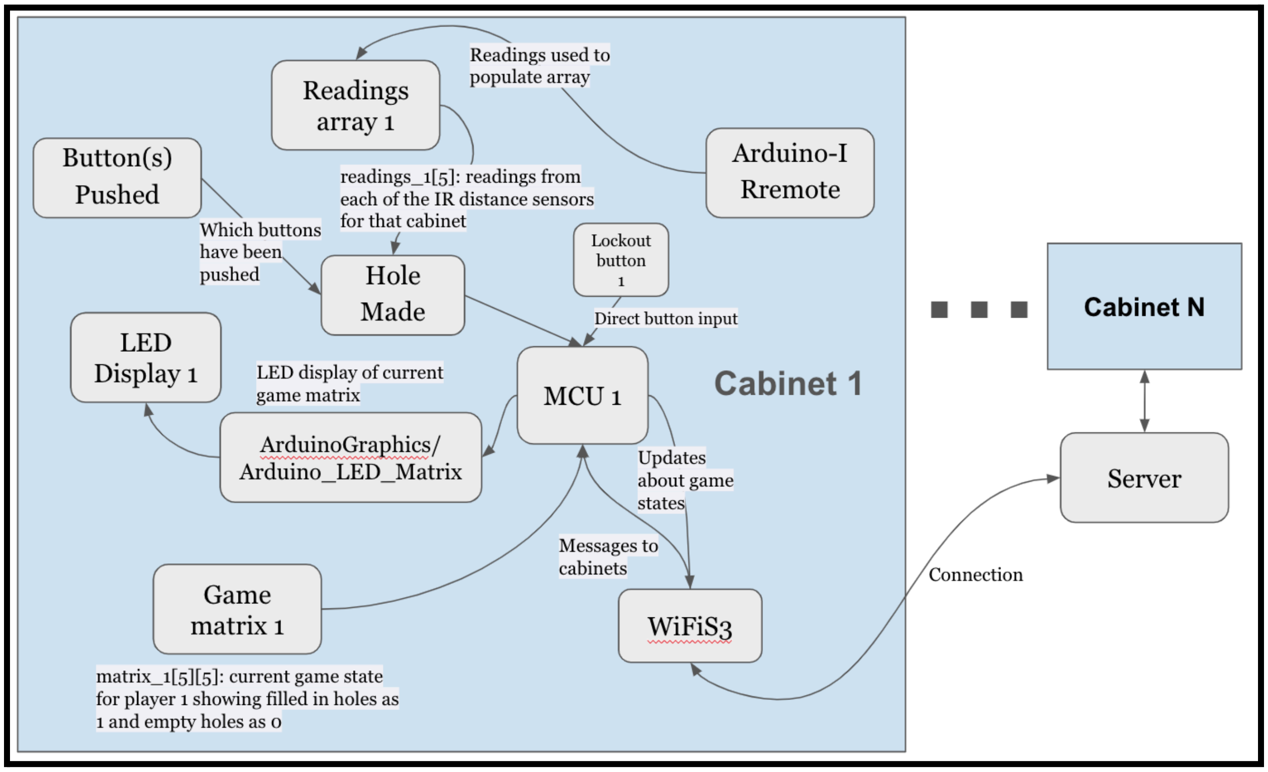
a. If a winning cabinet exists, the game manager **shall** reply to all future hole update or heartbeat requests from any connected cabinet with the ID of the winning cabinet, to indicate the game should be ended
b. If a winning cabinet does not exist, the game manager **shall** reply to all future hole update or heartbeat messages from any connected cabinet with a null ID, to indicate the game should continue

RO-M: A cabinet in its *game loop* **shall** send a heartbeat message to the game manager at a *regular interval*

RO-N: A cabinet, upon receiving an appropriate error code or correctly ending the game should transition into the *Reset state,* from which the cabinet will reenter the *Attempting state*

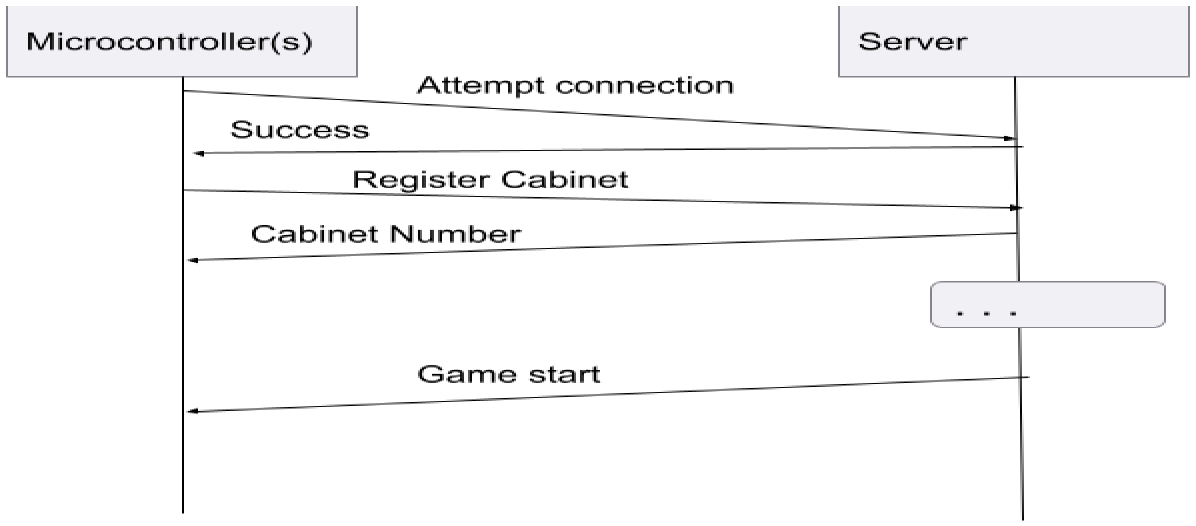
3. Revised architecture diagram from part 2 of the milestone report.


Cabinet 1
Board with buttons for each row
This same model is all for all connected cabinets
IR distance sensor for each column
Cabinet N
Onboard LED Board
Wired up
Golf ball
Connection via WiFi
Connected to some power source
Lockout button
Game Manager

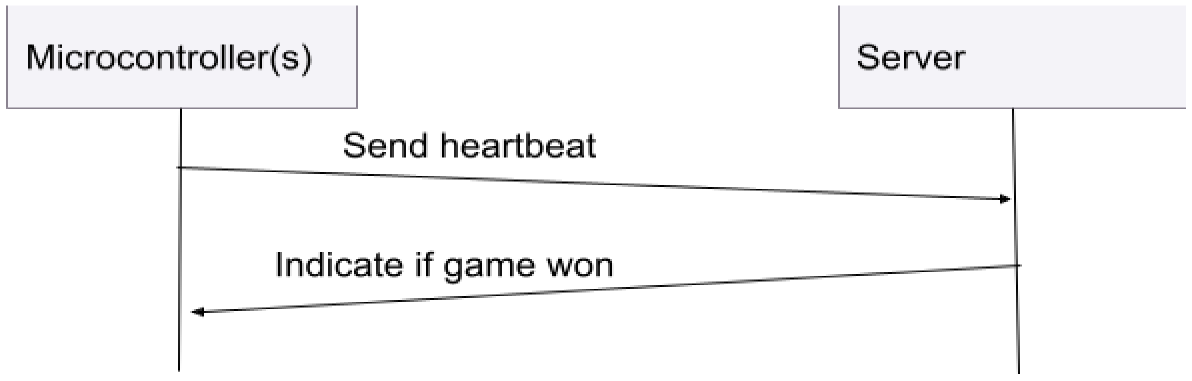

Readings array 1
Readings used to populate array
Arduino-I Rremote
Button(s) Pushed
readings_1[5]: readings from each of the IR distance sensors for that cabinet
Which buttons have been pushed
Hole Made
Lockout button 1
Direct button input
Cabinet N
LED Display 1
LED display of current game matrix
MCU 1
Cabinet 1
ArduinoGraphics/ Arduino_LED_Matrix
Updates about game states
Server
Messages to cabinets
Game matrix 1
Connection
WiFiS3
matrix_1[5][5]: current game state for player 1 showing filled in holes as 1 and empty holes as 0

4. ***Three sequence diagrams for reasonable use case scenarios of your system (if there are more than 3, you should pick the 3 most likely use case scenarios to diagram, and include a note as to what scenarios you left out). Before *each* diagram, you should write a short (1-2 sentence) description of what the use case scenario is.**

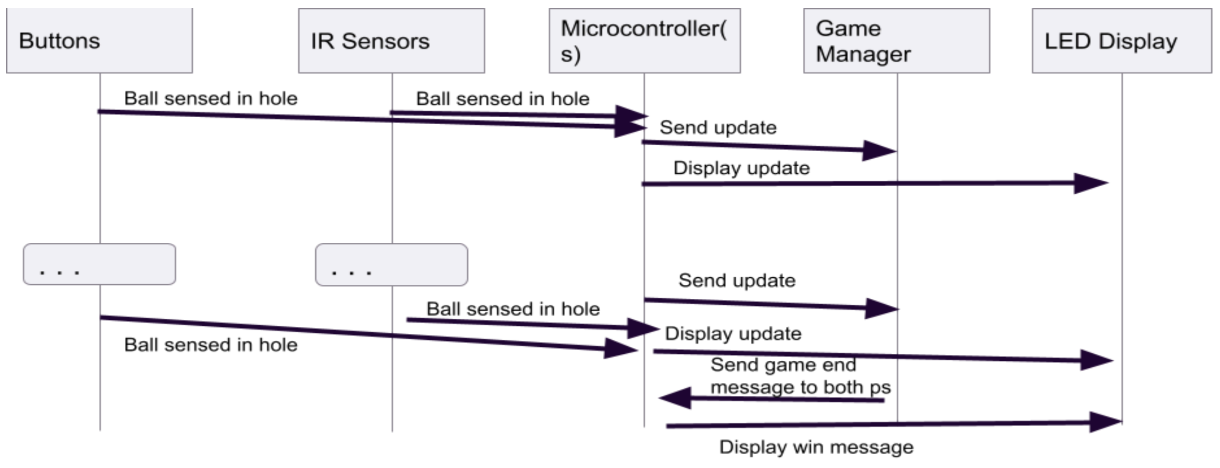
Sequence 1 (Startup): Initial connection between cabinets (microcontrollers) and the server, in addition to the game start message after connection.



Sequence 2 (Heartbeat): Represents the intermittent pinging of the server by the microcontroller both to assess that the server is still alive and whether the game has been won



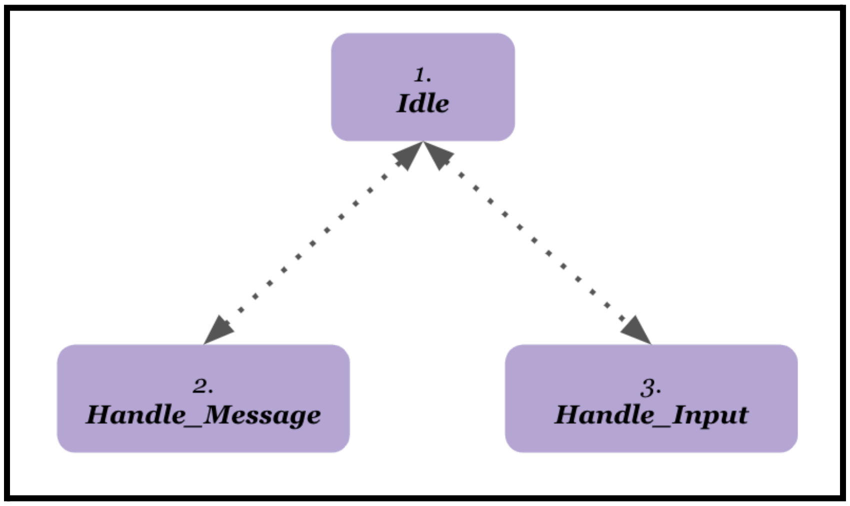
Sequence 3 (Hole detection): Represents the process of a golf ball having been detected in a certain hole by both the button-based pressure plates and the IR sensors, in both the case that the game is won as a result of the hole or that it is a standard non-winning entry.


Buttons
IR Sensors
Microcontroller(s)
Game Manager
LED Display
Ball sensed in hole
Ball sensed in hole
Send update
Display update
. . .
. . .
Send update
Ball sensed in hole
Ball sensed in hole
Display update
Send game end message to both ps
Display win message

5. **Revised finite and/or extended state machine from part 4 of the milestone report. See part 13 below for directions on including your peer review feedback as an appendix.**

Our manager was not strictly implemented adhering to an FSM model, as our code is not running on an embedded system. Instead, our server maintains multiple threads—one for processing messages, and another for handling input. We attempted to model this accordingly in the following FSM and transition table, but note that receiving messages from cabinets or keyboard input to our REPL is what triggers our transitions.

Game Manager FSM



Inputs:

- **keyInput:** MenuSelection{GAME_SETTINGS, START_GAME, LINE_PATTERN, BLACKOUT_PATTERN, RESET_ALL}
- **message:** Network request struct, with relevant fields:
    - **type**: MessageType{REGISTER_CABINET, REQUEST_START, HOLE_UPDATE, HEARTBEAT}
    - **id:** int
    - **body:** int

Variables:

- **ACTIVE_PATTERN**: GamePattern{BLACKOUT, LINE} = LINE
- **GAME_STARTED**: bool := false
- **CONNECTED_CABINETS**: dict[int, float[5][5]])
- **NEXT_CABINET**: int = 0

- **winningCabinet**: int = -1

Methods:

- **registerCabinet(int):**
    - Adds a new cabinet to **CONNECTED_CABINETS**, and increases **NEXT_CABINET** by 1
- **sendCabinetResponse(int):**
    - Sends a reply to the requesting cabinet with an ID, and registers the ID
- **printActivePattern(str):**
    - Prints the selected game pattern with a message
- **sendStartResponse(bool):**
    - Sends a reply to the requesting cabinet with the current status of GAME_STARTED
- **sendHeartbeatResponse(int):**
    - Sends a reply to the requesting cabinet with the ID of the current winning cabinet if there is one, otherwise replies with "N"
- **sendHoleUpdateResponse(int):**
    - Sends a reply to the requesting cabinet with the ID of the current winning cabinet if there is one, otherwise replies with "N" OR "Invalid cabinet" if the cabinet number is not known
- **checkWinning(int, GamePattern):**
    - Checks whether the provided cabinet ID has won based on the provided game pattern, and returns its ID if so, -1 otherwise

Game Manager Transition Table				
Transition	Guard	Explanation	Output	Variables
1-2(a)	message && message.type == REGISTER_CABINET	Waiting for a messages from clients – received cabinet connection	sendCabinetResponse(NEXT_CABINET)	registerCabinet(NEXT_CABINET)
1-2(b)	message && message.type == REQUEST_START	Waiting for a messages from clients – received request start	sendStartResponse(GAME_STARTED)	N/A
1-2(c)	message && message.type == HEARTBEAT	Waiting for a messages from clients –	sendHeartbeatResponse(winningCabinet)	N/A

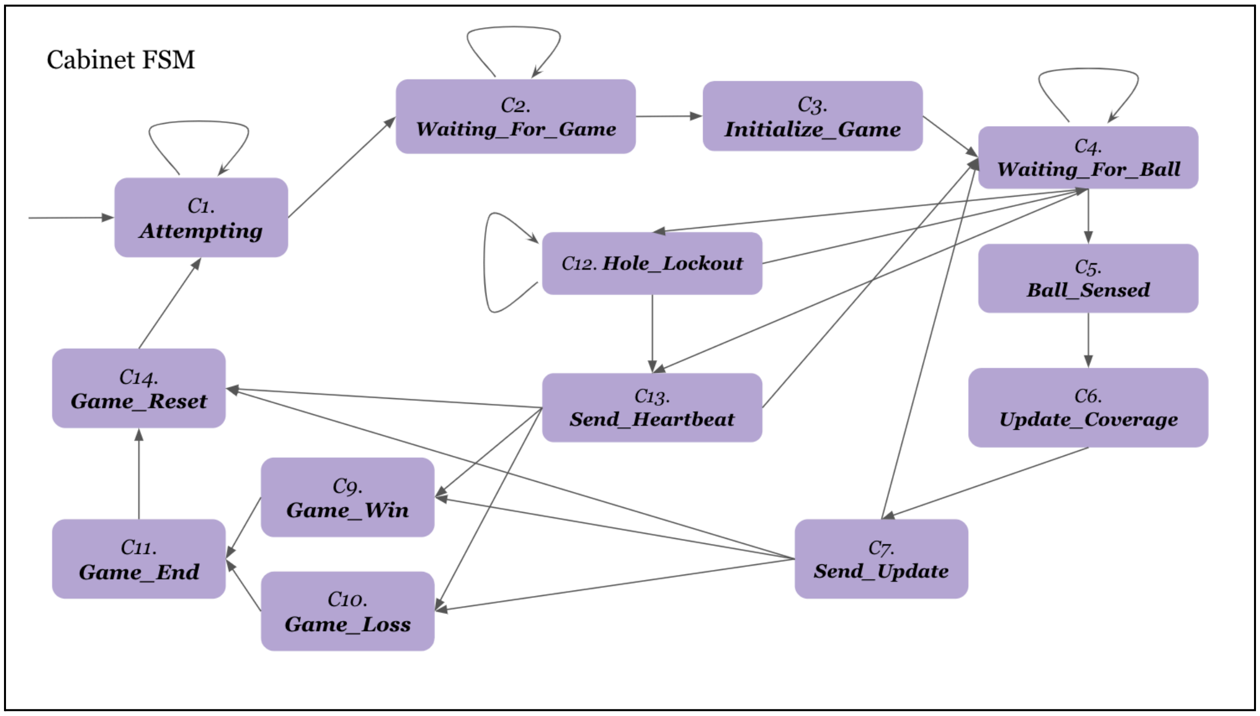
		received heartbeat		

1-2(d)	message && message.type == HOLE_UPDATE && winningCabinet == -1	Waiting for a messages from clients – received hole update and there is no winner	sendHoleUpdateResponse(winningCabinet)	CONNECTED_CABINETS[message.id][body] := 1
				
				winningCabinet := checkWinning(ACTIVE_PATTERN, message.id)

1-2(e)	message && message.type == HOLE_UPDATE && winningCabinet != -1	Waiting for a messages from clients – received hole update and there is a winner	sendHoleUpdateResponse(winningCabinet)	
1-3(a)	**keyInput** == BLACKOUT_PATTERN	Received input to update active pattern to BLACKOUT	printActivePattern("“Setting game pattern to {ACTIVE_PATTERN}!”)	ACTIVE_PATTERN := BLACKOUT
1-3(b)	**keyInput** == LINE_PATTERN	Received input to update active pattern to LINE	printActivePattern(“Setting game pattern to {ACTIVE_PATTERN}!”)	ACTIVE_PATTERN := LINE
1-3(c)	**keyInput** == START_GAME	Received input from REPL – starting game	N/A	GAME_STARTED := True

1-3(d)	**keyInput** == RESET_ALL	Received input from REPL – resetting all cabinets	N/A	NEXT_CABINET := 0
				winnerCabinet := -1
				GAME_STARTED := False

2-1	N/A	Return to idle state after response is sent	N/A	N/A
3-1	N/A	Return to idle state after input is handled	N/A	N/A



Inputs:

- **infraredSensors**: {IR_1, ..., IR_5}
- **rowButtons**: {B_{11}, B_{12}, B_{21}, B_{22}, ..., B_{51}, B_{52}}
- **lockoutButton**: L

Variables:

- **boardState:** uint32[3] = {0x0, 0x0, 0x0}
- **lockoutState**: uint32[3] = {0x0, 0x0, 0x0}
- **sensorReadings**: float[5][SENSOR_WINDOW + 2] = {DEFAULT_SENSOR_READING, ..., DEFAULT_SENSOR_READING, SENSOR_WINDOW * DEFAULT_SENSOR_READING, DEFAULT_SENSOR_READING}
- **activeRow**: int = -1
- **activeHole**: int = -1
- **lockedOut**: bool = false
- **lockoutSwap:** bool = false
- **lockoutCounter:** int = 0
- **heartbeatCounter:** Int
- **CABINET_NUMBER**: int = -1
- **matrix**: ArduinoLEDMatrix
- **Constants:** Variables that are held fixed during the program

    - DEFAULT_SENSOR_READING := 30
    - SENSOR_WINDOW := 15
    - lockoutThreshold := 50000
    - heartbeatThreshold := 50000
- **Transient Variables:** Local variables used only to represent computations; these do not persist between states, and should be assumed as being absent outside of the contexts in which they are assigned to
    - _SHR
    - _SHUR
    - _MATRIX_ROW
    - _MATRIX_COL

Methods:

- **Helpers**: Methods with a corresponding method in the project files (though their signatures may differ, as they may handle updates internally)
    - **checkHeartbeat()** → bool
        - Increments heartbeatCounter (mod heartbeatThreshold), and returns whether a heartbeat should be sent
    - **initializeSensors()** → float[5][SENSOR_WINDOW + 2]
        - Sets all **sensorReadings** *readings* [index 1 → SENSOR_WINDOW], *total* [index SENSOR_WINDOW + 1], and *average* [index SENSOR_WINDOW + 2] to their default values
    - **activateHole(float):**
        - Toggles an index in the LED matrix, and updates boardState accordingly
    - **checkShouldStart()** → {SUCCESS, FAILURE, ERROR}:
        - Network method that sends a game start request and returns whether a game should begin, send
    - **sendHeartbeat()** → int ≥ -2:
        - Network method that sends a heartbeat message and returns the cabinet ID of a winning cabinet
    - **sendHoleUpdate(int, int)**:
        - Provided CABINET_NUMBER and activeHole, sends a **hole update message** to the server
    - **updateSensorAverages()** → float[][]:
        - Method to maintain moving averages for IR sensor readings by polling all **infraredSensors**
    - **toggleLockoutPattern(bool)**:
        - Updates the R4 WiFi onboard LED display with the value of either boardState or lockoutState
    - **matrix.loadFrame(uint32[3]):**
        - Updates the R4 WiFi onboard LED display with the provided frame
    - **lockoutISR():**

        - An interrupt that is called whenever the **lockoutButton** is pressed in order to toggle the lockedOut variable back to false.
    - **setupServer():**
        - Sets up the connection with the server
    - **computeActiveColumn():**
        - Determines the active column based on the sensor readings
    - **displayMessage(str, int):**
        - Display the message on the onboard LED matrix
- **Pseudo-Helpers:** Methods that have corresponding *functionality* in the project files, but do not necessarily map to a specific helper function
    - **checkActiveRow()** → int:
        - Polls the all the **rowButtons** to check whether any have been activated and update the active row if so
    - **holeToBoard(int)** → int, int:
        - Turns a 1D index into an index to toggle an LED in boardState
    - **getCabinetNumber**() → Int
        - Returns the <u>cabinet number</u> that would be returned by setupServer()
    - **getShouldStart**() → Int
        - Returns the should start <u>reply</u> that would be returned by checkShouldStart()
    - **getHeartbeatReply()** → Int
        - Returns the <u>cabinet reply</u> that would be returned in sendHeartbeat()

Cabinet Transition Table				
Transition	**Guard**	**Explanation**	**Output**	**Variables**
1-1	CABINET_NUMBER == -1	If our server did not provide our cabinet with an ID to use, we self-loop back to keep trying again; the server might not be started or our Arduino might not be connected to WiFi yet	setupServer()	CABINET_NUMBER := getCabinetNumber()

1-2	CABINET_NUMBER != -1	Connection is made from game manager to our cabinet and now the system is waiting for the game to begin.	No output	No change
2-2	checkShouldStart() == FAILURE	Cabinet polls the server to see if the game has started and it return FAILURE	No output	No change
2-3	checkShouldStart() == SUCCESS	Cabinet polls the server to see if the game has started and it return SUCCESS	No output	No change
3-4	N/A	Game should begin; this state clears game board and sets up LED matrix	matrix.loadFrame(boardState)	No change

4-4	!lockedOut && (checkActiveRow() == -1) && (heartbeatCounter != 0)	"Typical" game sensing loop: updates sensor averages, polls for push button input, and updates heartbeat counter	No output	sensorReadings = updateSensorAverages()
				
				heartbeatCounter := (heartbeatCounter + 1) % heartbeatThreshold

4-5	!lockedOut && (**checkActiveRow() != -1)** && (heartbeatCounter != 0)	Ball has been sensed!	No output	sensorReadings = updateSensorAverages()
				
				activeRow := checkActiveRow()

4-12	**lockedOut**	Ball lockout is active, so sensors should not be updated, but a heartbeat or other check may have placed us back in WAITING_FOR_BALL—return back to HOLE_LOCKOUT	No output	No change
4-13	!lockedOut && (activeRow == -1) && **(heartbeatCounter == 0)**	Heartbeat counter has reached its threshold, and no ball was sensed	No output	sensorReadings = updateSensorAverages()
5-6	N/A	Computes which hole has achieved coverage, based on sensor data computed in WAITING_FOR_BALL	No output	activeHole = 5 * activeRow + computeActiveColumn()
6-7	N/A	Coverage is updated on the LED matrix	activateHole(activeHole)	_MATRIX_ROW, _MATRIX_COL := holeToBoard(activeHole) boardState[_MATRIX_ROW] |= (1 << (31 - _MATRIX_COL)

7-4	sendHoleUpdate(CABINET_NUMBER, activeHole) == -1	Return to normal game loop if nobody has won	No output	lockedOut := true
				
				lockoutState := boardState
				
				activeHole := -1
				
				activeRow := -1

				sensorReadings := initializeSensors()
7-9	sendHoleUpdate(CABINET_NUMBER, activeHole) == CABINET_NUMBER	Transition to win state if the server returns that the active cabinet has won	No output	No change
7-10	**_SHUR** := sendHoleUpdate(CABINET_NUMBER, activeHole) **_SHUR >= 0 && _SHUR != CABINET_NUMBER**	Transition to lose state if the server responds that a different cabinet has won	No output	No change
7-14	sendHoleUpdate(CABINET_NUMBER, activeHole) == -2	Reset game in event of bad response message	No output	No change
9-11	N/A	Game ends; you won!	displayMessage(“winner, winner, chicken dinner!”, 150)	No change
10-11	N/A	Game ends; you lost :(	displayMessage(“loser, loser, lemon snoozer!”, 150)	No change
11-14	N/A	After game end, reset and try again	No output	No change
12-4	!lockedOut	Transition back to WAITING_FOR_BAL	~	~

		L we are no longer in lockedOut		

12-12(a)	lockedOut && !checkHeartbeat() && lockoutCounter == 0	Update the flashing "L" on our display when the flash counter rolls over, and progress counter	toggleLockoutPattern(!lockoutSwap)	lockoutSwap := !lockoutSwap
				
				lockoutCounter := (lockoutCounter + 1) % lockoutThreshold

12-12(b)	lockedOut && !checkHeartbeat()	Progress our hole flash counter	No output	lockoutCounter := (lockoutCounter + 1) % lockoutThreshold
12-13	checkHeartbeat()	Transition to sendHeartbeat() in case we have met the threshold	No output	No change
13-4	sendHeartbeat() == -1	Return to normal game loop if nobody has won	No output	No change
13-9	sendHeartbeat() == CABINET_NUMBER	Transition to win state if the server returns that the active cabinet has won	No output	No change
13-10	**_SHR** := sendHeartbeat() **_SHR >= 0 && _SHR != CABINET_NUMBER**	Transition to lose state if the server responds that a different cabinet has won	No output	No change

13-14	sendHeartbeat() == -2	Heartbeat receives a connection failure; reset game	No output	No change

14-1	N/A	(Re)initialize all variables in order to have a fresh start for a new game	matrix.loadFrame({0x0, 0x0, 0x0})	activeRow := -1
				activeHole := -1
				
				heartbeatCounter := 1
				
				lockoutCounter := 0
				
				lockoutSwap := false
				
				lockedOut = false
				
				boardState := {0x0, 0x0, 0x0}
				
				sensorReadings := initializeSensors();

6. *Revised traceability matrix from part 5 of the milestone report.

Traceability Matrix

7. **An overview of your testing approach, including which modules you unit tested and which integration and system (software and user-facing/acceptance) tests you ran. You should explain how you determined how much testing was enough (i.e. what coverage or other criteria you used, how you measured that you achieved that criteria, and why you think it is sufficient for you to have met that criteria).**

Our project has two main components: the server code and the client code. We unit tested and integration tested both components separately and then system tested the whole game together. Our sequence diagrams guided our testing in that we wanted to ensure that each of them would be fully functional. It is denoted below where each of them was tested (usually in multiple locations).

Individual Testing:

In addition to the physical test cases we wrote, we also performed extensive live testing on the system at several stages throughout the project. In the beginning, we tested the WiFi connecting between the Arduinos and a central computer to ensure that our plan for the server-client model would work (*Sequence Diagram 1: cabinet connection*).

We also tested all of our different ideas for sensors by themselves as well as with our cabinets before we decided on which ones to move forward with. This included testing the ultrasonic sensors, the IR sensors, the copper tape, the piezo discs, and the button pressure plates. Each of these testing processes required us to write code to read in the values of the given sensor as well as convert that value to some useful metric for our system. In addition to just testing the functionality of the sensors, we also did rigorous testing on specific location of the sensors in our cabinet including height from the table, orientation of the sensor, location in regards to holes (centered vs offset), and distance from each other (*Sequence Diagram 3: sensors detecting ball*).

Unit Testing:

For the server code, we unit tested any functions used that were not directly related to communication with the clients (server tests 0 to 7). One such function was Cabinet.did_win() which checks if the cabinet has won (completed the indicated GamePattern). To test this function, we tested all of the possible cases: (1) cabinet won with a vertical 5-in-a-row; (2) cabinet won with a horizontal 5-in-a-row; (3) cabinet won with an up diagonal 5-in-a-row; (4) cabinet won with a down diagonal 5-in-a-row; (5) cabinet won with a blackout; (6) cabinet did not win for 5-in-a-row; (7) cabinet did not win for blackout. Another function we unit tested Cabinet.update_hole() in a similar manner. This function is much simpler and only requires one unit test to ensure that the intended hole is being set (*Sequence Diagram 3: testing for win*).

For the client side, we tested the FSM as we did in lab 6 by creating mock functions and testing each transition. The Google Sheet linked below shows all of the transitions we tested and what the variables were for that transition (*Sequence Diagrams 2 and 3: functionality at states SEND_HEARTBEAT, BALL_SENSED, UPDATE_COVERAGE, SEND_UPDATE*).

final_tests

Note: We did not directly test the server FSM in the same way we did the client FSM since the server code does not follow the same natural flow.

Integration Testing:

Our integration testing consists primarily of mock functions for the server in order to test its functionality in a more isolated way. We test certain server functions (do_GET, do_POST, handle_register_cabinet, handle_request_start, handle_hole_update, sendHeartbeat) based on the mock responses from clients (server tests 8 to 18). We tested each possible message from the client in each of these functions and ensured that (1) the expected response from the server is sent and (2) the variables are set to the expected values (*Sequence Diagram 2 and 3: functionality of sendHeartbeat and handle_hole_update*). Additionally, we tested the server REPL for each of the expected user inputs (server tests 19 to 21). We again checked that the proper variables are set when doing this (*Sequence Diagram 1: game is successfully started*).

We also test the client communication with the server through mock functionality in certain client functions, such as sendHeartbeat(), where we mimic certain server responses and test the client functionality. These tests are woven in with our FSM units tests from above. These tests were not particularly guided by our sequence diagrams.

System Testing:

In order to do system testing, we just played our game from start to finish. This allowed us to test the client-server connection and communication, the reliability of our sensors, and our FSM functionality. This also gave us an idea of what the user was thinking and feeling as they are playing the game. Any issues we encountered here, we were able to attribute to some new variable in the system since all of our unit and integration tests were successful. Examples of these problems included using a different WiFi network or trying to use a different power source (*Sequence Diagrams 1, 2, and 3: all 3 occur during every game played*).

Through this system testing, we verified the functionality of certain client functions through game play which are used in the FSM such as clearBoardState(), activateHole(), enableLED(), refreshLockoutPattern(), and toggleLockoutPattern(). Due to how they are woven into the flow of the game, they were difficult to unit test.

Additionally, we system tested our watchdog timer. We used the built in Arduino WDT library in order to implement our watchdog. Because of this, we could not unit test our watchdog, so we system tested it by ensuring that it will "bark" if there is a delay in starting the next loop of more than 5 seconds and that it will not "bark" if there is normal functionality. This delay would be caused by a problem with one of the networking functions if the client cannot connect to the server causing the code to hang.

One piece of feedback we got about our testing approach from our milestone report was that we need to further detail how to evaluate how much testing is enough. As detailed above, we did rather extensive testing on each of the components in our system. We started with the individual testing and once we were confident with the results we moved onto the next stage. To us, “confident” in terms of the sensors meant that it produced the correct hole the majority of the time (although we didn’t have an exact figure in mind for this). From a coverage perspective, this meant comprehensively testing that each piece of hardware, being the IR sensors and buttons, reported correct responses consistently. In terms of the WiFi connection, “confident” meant that it connected and communicated correctly every time. We put more emphasis on the WiFi than the sensors since the game cannot be played without successful connection between client and server. In terms of the system testing, we wanted to ensure users could successfully play a game from start to finish. We knew we did enough testing when we could successfully play the game several times in a row. Additionally, for our FSM testing, we knew we did enough when every transition was covered by our tests (even the transitions with no guards). For our server testing, we knew we did enough when every possible combination of wins was tested and when we had tested all of the possible messages from the client. For our game as a whole, we also strived to cover each of the possible sequence diagrams in our testing (as detailed above).

8. **At least 2 safety and 3 liveness requirements, written in propositional or linear temporal logic, for your FSM. Include a description in prose of what the requirement represents. Each requirement should be a single logic statement, made up of propositional and/or linear temporal logic operators, that references only the inputs, outputs, variables, and states defined in part 5 of this report. We will discuss safety and liveness requirements on the week of November 27th. You can also refer to chapter 13 of Lee/Seshia or chapters 3 and 9 of Alur's textbook (Brown login required).**

Liveness #1:

- **G(Ball_Sensed → F(Hole_Lockout ∨ Game_End))**
    - This property ensures that when a ball is sensed, the cabinet always enters either the hole_lockout state if the game is not won as a result of that hole, or the Game_End state if the game is one as a result of that hole.

Liveness #2:

- **G((Game_Win ∨ Game_Loss) → F(Attempting))**
    - This property represents that after a game is won or lost the cabinet returns to the attempting state so that a new game can begin.

Liveness #3:

- **G(If ¬(connected) → F(Listening))**
    - This property ensures that if one or both cabinets disconnects mid game, the server will always attempt to reset and reconnect.

Safety #1:

- **G(Initialize_Game → X(Waiting_For_Ball))**
    - This property ensures that when the cabinet has been notified that a game has been initialized, it immediately enters the Waiting_For_Ball state.

Safety #2:

- **¬(FG(Game_Reset)**
    - This ensures that in a single state, the game is never stuck in the Game_Reset state.

9. **A discussion of what environment process(es) you would have to model so that you could compose your FSM with the models of the environment to create a closed system for analysis. Composing state machines was covered during the modeling lectures. For each environmental process, you should explain and justify whether it makes sense to model it as either a discrete or hybrid system *and* either a deterministic or non-deterministic system (for example, a button push could be modeled as a discrete, non-deterministic signal because it is an event that could happen at an arbitrary time, but a component's position could be modeled as a hybrid, deterministic signal based on some acceleration command). You are not being asked to actually do this modeling or any sort of reachability analysis, just to reason about this sort of modeling at a high level.**

We have three main environmental processes that would help compose our FSM and make it a closed system. These are the reset buttons (which is the button to get you out of lockout state), the column reading (which comes from the ir sensor reading), and the row reading (which comes from the row button/pressure plates input). If we are able to successfully model the environment and align it with our software/FSM, we can create the closed system we are looking for.

For the reset buttons, they would compose a discrete, non-deterministic system. They are discrete because they can be pressed at any point in the game and the button really only has two states (pressed or unpressed). The system is non-deterministic because there are multiple possible next states based on the pressed or unpressed input. For example, the button can be pressed when in lockout or when not so it could go from locked to unlocked or from unlocked to unlocked.

For the column inputs/sensor readings, these are hybrid systems. The sensors are always reading in input and because of this continuous input (that changes based on objects being in front of it or not), it is a hybrid system because it has the continuous behavior of reading in a distance and the discrete behavior of reading in any distance at an arbitrary time. The idea also is that the sensor is either sensing an object (the golf ball) or not sensing the object. The system is non-deterministic because there are multiple iterations of the next possible state because this is dependent on where the human input rolls/drops the ball. However, because it is always polling, the system as a whole always has the same next state of just waiting for a button input. The button input is what determines what is done with the readings but the readings continue regardless of the other components in the game.

The final process is the row inputs or pressure plate buttons. These are discrete in the same way the reset button is discrete. This is also a non deterministic system. There are multiple possible next states based on the either pressed or unpressed input. Additionally, due to the pressure plate having the hardware redundancy of two buttons per pressure plate, there are more combinations of button pushes that result in the various next states. For example, buttons one, two, or one and two can trigger the same next state so we have to account for this when modeling the environment.

Ultimately, when we consider all of these environment processes, we are able to mock them in our testing and in conjunction with our FSM, we can create a closed system for testing. Inputs are external to the system so they are the environment in which the system operates. By modeling these, we can reduce our testing to focus on the transitions between states and prepare for various unexpected changes in the environment.

10. **A description of the files you turned in for the code deliverable. Each file should have a high-level (1-3 sentence summary) description. For each of the assignment requirements (PWM, interrupts, serial, etc), you should indicate which file(s) and line(s) of code fulfill that requirement.**

The high-level structure of our code is split based on cabinet files (.ino, .h) which are in the top-level directory, and server files in the local-server directory:

- **Clamball.ino**
    - This is the primary manager for cabinet logic; the file contains all variables used by our FSM, manages our state updates, handles sensor reads, and defines all of the non-networking code used for our game
    - **ADC:**
        - We utilize a suite of 5 infrared distance sensors (specifically, Sharp GP2Y0A21) to identify which column the game ball falls into, based on an average of the readings taken
        - Our sensor data sheet provides a map of voltage to distance readings; however, rather than attempt to derive the non-linear relationship ourselves, we simply used online reference code for working with these sensors; the used conversion logic lives in *updateSensorAverages* (~510–542)
    - **ISR:**
        - Our cabinets define digital pin 3 as the LOCKOUT_PIN, and attach *lockoutISR* (~208–214) as the ISR that gets called when the lockout button on the cabinet is pressed. This indirectly returns the cabinet to "normal" operation (e.g. WAITING_FOR_GAME) from in HOLE_LOCKOUT mode, allowing the user to reach into the cabinet without interfering with game logic prior to pressing the button
    - **Watchdog:**
        - Our watchdog is configured with a 5s window, and is configured using *setupWdt* (~236; see **networking.ino**). Additionally, our WDT is *very* coarse grained: we pet the watchdog at every iteration of our FSM loop, as the only aspect that can "hang" is our various network requests. While we experimented with book-keeping (i.e. petting in tighter space), we found it was clearer to expose the watchdog logic as part of the broader FSM function (~297)
    - **WiFi:**
        - Various functions which make WiFi requests (defined in **networking.ino**) are called at points within the FSM: *setupWifi* in *setup* (~232), *setupServer* in the ATTEMPTING state (~315), *checkShouldStart* in WAITING_FOR_GAME (~325), *sendHoleUpdate* (~400), and *sendHeartbeat* (~472)
- **networking.ino**
    - **Watchdog**

        - Our WDT is configured using a 5s interval (~71–80), unless it fails to initialize, in which case we simply enter a loop**.**
    - **WiFi:**
        - WiFi communications between client and server are managed by functions defined across this file. The credentials specified in **network_config.h** are used in *setupWifi* (~13–35) to create a WiFi connection, and an HTTP connection to the requested server IP/port in *setupServer* (~37–69).
        - Get and post requests to various server endpoints (/register-cabinet @ ~46, /request-start @ ~85, /heartbeat @ ~149, /hole-update @ ~129) are abstracted out into helper methods which return a usable response to the cabinet FSM, allowing networking code to be handled independent of business logic
- **network_config.h:**
    - **WiFi:**
        - This file is used to defined various properties of the WiFi network and server connection that an Arduino needs to use (WiFi being open, SSID, password if necessary, as well as the IP/port of the game manager server); this metadata is used by **networking.ino**, as well as ensuring "local" configuration does not get committed to the repo
- **local-server/server.py**
    - This file defines and runs a local server to manage our *game manager*, whose operation is detailed in an FSM above. Our server is able to manage the initiation of games, define the game pattern, and service WiFi messages sent from clients
- **local-server/utils.py**
    - Utilities for outputting printing colored text (e.g. for test results); repurposed from Zack's personal project
- **local-server/tests.py**
    - Server unit tests!

11. **A procedure on how to run your unit tests (for example, if you have mock functions that are used by setting a macro, similar to lab 6, make sure to note that).**

We have two different testing files as there are two different sides to our system: the server and the clients.

Testing server:

In order to run the unit tests for the server, first ensure that "`TESTING = True`" is uncommented at the top of *server.py*. Then, run the python file *local-server/tests.py* in your terminal. This will run all of the unit tests for the server. The output of this will be each of the tests followed by PASS or FAIL to indicate if the test has passed. The details about what each of these tests are doing is described in *local-server/tests.py* in a comment above each test.

Testing client:

In order to run the unit tests for the client functions and FSM, ensure that the line "`#define TESTING`" is uncommented and "`networked`" is set to "`false`" at the top *Clamball.ino*. Then make sure to reupload the code to your Arduino. This will run the tests in *Clamball_tests.ino* which test each possible transition in our FSM. The transitions tested are shown below with each of the variables used to test the given transition. Similarly to above, the output of this will be each of the tests followed by PASS or FAIL to indicate if the test has passed.

final_tests

12. **A reflection on whether your goals were met and what challenges you encountered to meet these goals. You should only include challenges met or newly addressed since the milestone.**

Our proposed goals were to implement a 2-player, networked version of (not-so) “mini” Fascination. While we ran into *many* challenges in the process, we were ultimately able to accomplish this goal.

The main challenge we faced throughout the *entire* design process was sensor reliability. We dismissed early on the “ugly” but functional approach of having a unique sensor for *every* hole—ambiguously: a pressure, toggle, or button of some sort, mapped to an ISR—but felt this to be overkill and inelegant (not to mention: expensive, space-inefficient, and wire-dense). Hence, out of desire not to wire up 50 individual sensors, we decided to use ultrasonic distance sensors (HC-SR04s), having each sensor manage a column of holes, and bucketing the distance readings to determine which hole a ball fell into.

Looking back, we should have spent more time prototyping and researching sensors. We landed on HC-SR04s as a result of their cost and prior experience with them as a group, but failed to adequately investigate their limitations. Almost immediately, we realized these sensors did not have the fidelity to sense an object as small as a marble. Unfortunately, rather than change sensors, we simply scaled up our design without ensuring the ultrasonics would work even with the larger design. We quickly ran into massive interference issues: the wide detection angle of the sensors resulted in readings being recorded from *multiple* sensors, despite being in different columns. Additionally, the slow time per reading meant that, at most, 2–3 readings could be taken of the ball as it fell before it was out of range. We attempted to rectify this by placing walls between the sensors…only to realize that our sensors were always detecting the walls and nothing else.

Our second iteration was similar in concept, switching out ultrasonic sensors for infrared sensors. While prototyping with Sharp IR sensors in the BDW, we found both the wide beam and limited number of readings to largely be non-issues—the detection angle of the IRs was incredibly narrow by comparison, and they operated at (roughly) the speed of light, allowing dozens of readings in the same interval. However, we learned only after switching to use these sensors (an expensive swap, costing almost $100) that the sensors had incredibly high fidelity at short range, but appeared to have far noisier readings further away. This resulted in very consistent readings for nearby holes, but much lower reliability on detecting balls in the final 2 holes in a column.

Our third iteration was a homogenous sensor approach: if we could rely on the IR sensors to *mostly* determine the column, we could use a second method to detect the row a ball fell in, as the combination of row-column uniquely defines a hole. We were not, however, as confident on how to do this. We attempted to use copper tape, taping strips corresponding with rows of holes to a ramp under the cabinet, and suspending a taut second piece such that a falling ball would cause the two pieces to make contact. This approach, while simple in concept, was very fragile: as successive balls made contact with the tape, the top piece would lose its tension, sagging and making the contact switch unreliable. We pivoted to experimenting with piezo discs as force sensors, break-beam sensors, and other approaches, but with the time and (lack of) budget we had remaining for the project, we needed a reliable, cheap approach that was less subject to fast degradation.

Our final iteration, then, was another homogenous sensor approach, but using buttons rather than copper tape. We swapped our strips for "pressure plates," a thin piece of wood measured to be roughly the dimensions of a row of the cabinet, with two large push buttons hot-glued to the bottom side. The large surface area and multiple buttons means that a falling ball is effectively guaranteed to trigger at least one of the buttons in the row. We switched our to having buttons act as our "primary" sensors—spurious button presses are significantly less likely than noisy sensor readings—and switched our IR sensors to instead maintain a moving average (using an experimentally-determined window size of 15 readings for consistency) to smooth out the readings and mitigate random fluctuations. When a ball is detected as hitting a plate, we take the minimum average sensor reading, and assume that corresponds to the column. This combination of detection methods wound up being reliable enough to be able to determine which hole the ball fell into with *close* to 100% accuracy.

All in all: we all learned *many* valuable lessons—about woodworking, prototyping and research, sensor choice, about soldering (and how certain group members are not as skilled at it), hot glue as a more effective form of wood glue, how to kill your darlings (pour one out for Lucy & Mikayla's beautiful ramps), how to repurpose 50 drill press discs into makeshift trophies, and so much more. But, in the end: we did create 2 (mostly) functional ~~Fascination~~ Clamball cabinets, and even learned a little about embedded systems along the way :)

13. **As an appendix, include the review spreadsheets you received after the milestone demo. Each defect should be marked as "fixed" or "will not fix" with a justification.**

The linked spreadsheet below collects our received peer feedback; rows with **red** text in the fixed column are **won't fix**, as the feedback is either incorrect or not the intended behavior. **Yellow** text is also won't fix, but due to no longer being relevant as a result of structural changes between the milestone and final iteration of the project (e.g. switching from distance sensors as the primary to secondary detection method meant eliminating the 4-5 transition, hence feedback about those is no longer relevant). **Green** feedback has been explicitly incorporated or was naturally addressed by changes made over the design process.

#	Fixed?	Response	Transition or State #	Defect
1	🟥	This feedback is incorrect; an initial state arrow was present on the initial FSM	S1	Start state is indicated: Technically not explicitly shown, but can be easily inferred.
2	🟩		S4	I'm slightly confused about what the game manager is doing in this state. Is this the same REPL as state 2 (orchestration)? Why does the user need to send input to the game manager after the game has started?
3	🟩		S5	The description from the transitions out of this state seem to imply that the game manager sends a message and reads some data from the cabinets that indicate whether the game has ended. I think this should be more clearly stated in the table.
6	🟩	Our guards were poorly written for these states; GAME_WIN/LOSS →	8-11, 9-11, 10-11	Are all three states necessary for this FSM? Your guards are defined exactly the same

	🟩	GAME_END no longer have guards		
7	🟩		All	Inputs, outputs, and variables are defined: No inputs/outputs/variables are shown on the diagram itself
8	🟩		All	Input and variable initialization is included: diagram does not show which variables need to be initialized and which variables are input
9	🟩		All	Only inputs and variables are checked in guard conditions: No guards are explicitly shown
10	🟩		All	Only outputs and variables are set in actions: No variable/outputs are shown in the diagram
12	🟩		C11	variables are not re-initialized, so based on the current FSM, led_mat1 and led_mat2 will never satisfy the guard to take the transition C11-C2
14	🟩		C12-C1	variables are not re-initialized
18	🟩		Cabinet FSM	Confused about organization of the FSM, as in why some of the transitions don't have boolean operators between them
19	🟩		Cabinet FSM 12-8	For example, why do the withdrawal/error operations have boolean operators but the win operators do not
21	🟩		Cabinet FSM 8/9/10-11	Why are these different states if the guards and outputs are the same?
22	🟩		Game Manager FSM Overall	Little bit confused as to why guards on Game manager FSM are not defined variables
23	🟩		Game Manager Transition Table	Can you define the variables for the guards? How will you be keeping track of messages,

	🟩			user input, if a message is done sending, etc?
24	🟩		Interplay between two FSMs	At which state in the Cabinet FSM is it sending messages to the Game Manager FSM?
26	🟩		N/A	When are the win/lose or game_over_messages being sent to the game manager? I think this can be more clearly stated as transition outputs of the cabinet FSM.
27	🟩			In general, it seems like the FSM also needs to include a list of variables, the guard for every transition, what the output of each transition is, and what variables are modified at every transition. I can tell the FSM is very thought out and the table probably has all the information, but for this feedback form you don't have any of this information on the diagram itself.
4	🟨	These guards no longer have this structure	1-1, 1-2, 2-2, 3-4, 11-2	What is the operator between the conditions? &&? ||?
5	🟨	We no longer have a detection loop for our distance sensors	4-5, 5-4	Should there be an || between the two inputs?
11	🟨		C1-C2	I presume that Cabs2Server == True is part of the guard, but I'm not sure. Otherwise, this transition is not mutually exclusive from the C1-C1 transition.
13	🟨		C11-C2	t=2min shouldn't be in the guard
15	🟨		C2-C3	I'm not sure why Cabs2Server is being set to True here. Shouldn't the cabinets have

	🟨			already connected to the server when the cabinets connected to the game manager? I think another variable should represent the cabinets waiting for the game_start_message.
16	🟨		C2-C3	You can probably just initialize the timer variable when taking this transition.
17	🟨		C4-C12	There should be some guard here to have mutual exclusivity between this transition and C4-C5
20	🟨	~	Cabinet FSM 4-5/5-4	Why are the guards for these transitions the same? How is the variable change for 4-5 the same as guard for it
25	🟨	~	N/A	the sense_counter is never initialized in the FSM table