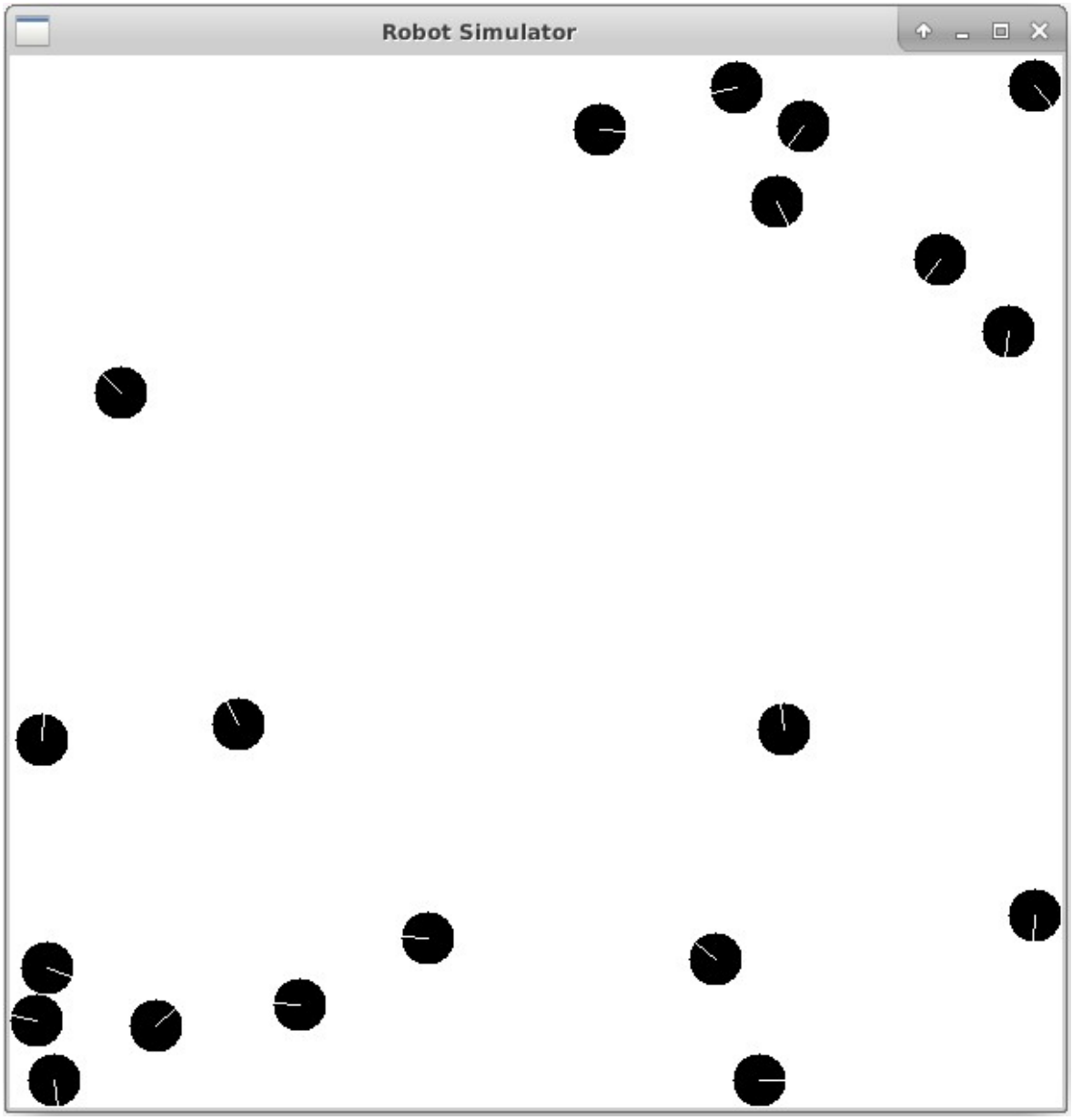
COMP2401 - Assignment #5

(Due: **Monday, November 30th, 2020 @ 6pm**)

In this assignment, you will make a simulator for simple robots that uses multiple threads and allows multiple robots to connect to it … with each robot running as its own process.



To begin this assignment, you should download the following files:

- **makefile** – use this to compile
- **simulator.h** – contains definitions and structs that will be used throughout your code
- **environmentServer.c** – contains a template for the code that will run the environment server
- **display.c** – contains the window/drawing code that you will use to display the robots
- **stop.c** – contains a template for the code for a process that will stop the simulator
- **robotClient.c** – contains a template for the code for a process that will run a single robot

When compiling the files, you will need to include the `-lm -lpthread` and `-lX11` libraries. (but the `-lX11` library (which is "minus L X eleven") is only needed for the **environmentServer.c** file since it uses a window and graphics.

Follow the steps below (in order) to complete the assignment:

(1) Examine the **simulator.h** file. The robots have a radius of **ROBOT_RADIUS**. Each robot is represented as an (**x**,**y**) location and a direction. The **x** and **y** values are the location of the robot in the window … which will be in the range from **ROBOT_RADIUS** to (**ENV_SIZE** - **ROBOT_RADIUS**). The **direction** should always be in the range from **-180°** to **+180°**. Each time a robot moves forward, it moves **ROBOT_SPEED** pixels in the direction that it is facing. Each time a robot turns, it turns **±ROBOT_TURN_ANGLE** degrees. The **Environment** contains up to **MAX_ROBOTS** robots in an array. **numRobots** is the number of robots currently registered with the server. The **shutDown** flag indicates whether or not the environment has been shut down, it must be set to **0** upon startup.

(2) The **environmentServer.c** file contains the code for the server template. The main function in this file MUST spawn two threads … one that will repeatedly accept incoming client requests … and another that will repeatedly redraw the robots in the window. Both of these threads should continue until the **shutDown** flag in the **Environment** has been set and all registered robotClients have been informed of the shutdown … at which point the threads will each exit gracefully. A pointer to the **Environment** should be passed in when spawning the threads so that the threads have access to the robots and the **shutDown** flag.

Write the code so that the two threads call these two functions, respectively:

```
void *handleIncomingRequests(void *environment)
void *redraw(void *environment)
```

The **redraw()** function is in the **display.c** file and has been completed for you. You MUST NOT alter any code in the **display.c** file. When you compile and run the **environmentServer.c** file, you should see a window come up which is empty. Run it in the background (i.e., use **&**) and then type **ps** in the terminal window to see the process id. When you close the window, you should see some XIO error (don't worry about this). Then typing **ps**, you should see that the environmentServer is no longer running.

Write the code in the **handleIncomingRequests()** function so that it starts up the server and repeatedly waits for incoming client requests. There are four possible incoming requests defined by the definitions in the **simulator.h** file: (1) **REGISTER**, (2) **STOP**, (3) **CHECK_COLLISION** and (4) **STATUS_UPDATE**,. For now, write code that accepts an incoming request to **STOP** the environmentServer. The code MUST be based on the UDP version of the client/server sockets that were discussed in the notes (i.e., not TCP). The incoming command for **STOP** should come in as a single byte. Upon receiving it, the server should shut down gracefully and both threads should be stopped cleanly.

Complete the code in the **stop.c** file so that it attempts to connect to the server and send the **STOP** command to the server. Test your code by running the **environmentServer** in the background and then running the **stop** program. If all works well, the environmentServer window should close and should shut down gracefully. Use **ps** to make sure that the **environmentServer** has indeed shut down properly as well as the **stop** program. Make sure that you don't have any segmentation faults.

(3) Now add code to the **robotClient.c** file so that it attempts to register with the running **environmentServer**. To do this, it should send a REGISTER command byte . It should then wait for a response from the server that should contain an **OK** or **NOT_OK** byte response. If the response byte was **OK**, then additional bytes should be received containing the id of the robot (i.e., its number in the environment array of robots), a randomly chosen (x, y) position and a randomly chosen direction. Note that it is the **environmentServer** that should choose the random location and direction (see step (1) above for valid ranges). If you are setting up the send/receive buffer as unsigned bytes, you will need to split the x, y and direction values into high and low bytes. You may also want to have an extra byte to indicate the sign of the direction (i.e., positive or negative) since the magnitude will be 0 to 180. You will need to adjust the **handleIncomingRequests()** function in the **environmentServer.c** file so that it sends the appropriate bytes back to the client. When all is working, you should see the robot appear in the simulator window. You'll need to run the **environmentServer** first (in the background) and then run the **robotClient** process (also in the background). Once it works, try running a second and third **robotClient** process … you should see 2 and then 3 robots appearing. Make sure that the **stop** process still shuts down the server properly.

(4) There is a limit to how many robots that can be added. It is set as **MAX_ROBOTS**, which is 20 by default. Adjust the code in the simulator server to deny any registrations that go beyond this limit. Simply send a **NOT_OK** response when it is full to capacity. Make sure that the **robotClient** handles this response properly. Test everything by adding 20 robots and then try adding a 21st robot. Make sure that you display an appropriate error message in the **robotClient** code so that it is clear when the client is unable to register. Don't test this yet until you get the automatic shutdown of your **robotClients** because otherwise you will need to kill 20 processes manually!!

(5) Now we will add functionality to the **robotClient** so that it repeatedly moves the robot around in the environment indefinitely. To move forward, the robot should first ensure that if it moves forward, it will not collide with any other robots or the environment boundary. To do this, you should send a **CHECK_COLLISION** request to the server which should contain the following information: the **CHECK_COLLISION** command, the robot's ID, the current location and direction of the robot. Keep in mind that if you use unsigned char array to send the bytes, you will need to break down the **x**, **y** into two bytes (most significant byte and least significant byte). You will also need to be careful sending the direction (which is ±180°) since an unsigned char only stores values in the 0-255 range and a signed char in the -128 to +127 range.

The robot should receive a single byte back from the server which has one of 4 values:

1. **OK** if the robot can move to that location without problems,
2. **NOT_OK_BOUNDARY** if the robot cannot move to that location because it would go outside the boundary of the environment, and
3. **NOT_OK_COLLIDE** if the robot cannot move to that location because it would collide with another robot.
4. **LOST_CONTACT** if the server is trying to shut down and the robot client needs to quit.

If all is **OK**, the robot should move forward by calculating a new location based on its current location and direction as follows: **(newX, newY) = (x+S*cos(d), y+S*sin(d))** where **(x,y)** is the current robot location, **S** is the robot's speed (in pixels) and **d** is the robot's direction. Otherwise, the robot should turn left or right (chosen randomly) by ROBOT_TURN_ANGLE degrees. To do this, do not change the (x,y) location, just change the direction. Make sure that the direction always remains in the ±180° range. Also, set your code up so that when the robot first hits a boundary or collides with another robot, it computes the random direction to

turn towards (CW or CCW). If the robot is unable to move on any successive turns, it should continue to turn in the SAME direction again. It should keep turning ROBOT_TURN_ANGLE degrees once per loop iteration. Only when it is OK to move forward should it begin its forward movement again. Coding things in this manner will ensure that the robot does not alternate turning back and forth in what appears to be an indecisive pattern. If coded properly, you should see the robot turning little by little upon collision.

(6) Add code to the **robotClient** so that it repeatedly sends **STATUS_UPDATE** messages to the server. These messages just send the latest robot location and direction so that the server knows where the robot is at all times. This allows the server to display it and is also necessary for collision detection later on. There should be no response back from the server when a **STATUS_UPDATE** is sent.

(7) You will notice after stopping, that some **robotClient** processes will still be running. You should kill these processes. You need to adjust your code so that all **robotClient** processes shut down automatically when the **environmentServer** shuts down. To do this, when the **environmentServer** gets a STOP request … it should not stop right away, but instead it should start informing the registered **robotClients** that the server is about to shut down. Since the **robotClients** are not in a loop listening for messages from the server, you will want to send a **LOST_CONTACT** response to each **robotClient** when it does a **CHECK_COLLISION** request. When a **robotClient** received such a response, it should shut itself down.

(8) To make things work, you will need to adjust the **handleIncomingRequests()** function in the **environmentServer.c** file so that it handles incoming **CHECK_COLLISION** requests from the robots. It will need to receive all the incoming data that was sent to it as part of the **CHECK_COLLISION** request. Write a function that determines whether or not the robot can move forward from the given location without collision. It will need to check the boundary values as well as the locations of all other robots to decide whether to return a value of **OK**, **NOT_OK_BOUNDARY** or **NOT_OK_COLLIDE**. You will also need to handle incoming STATUS_UPDATE requests so that you know (and update) the position of all robots at all times. Do NOT alter the **display.c** file, nor the **simulator.h** file.

(9) Test your code with a single robot to see if it works properly. You can lengthen the **usleep()** delays to slow things down and investigate the movements, but you must put them back to their initial values when things seem to be working properly. Test everything with multiple robots and ensure that the moving and collisions are working properly. You might end up with robots stuck on each other … see the next part as a possible explanation,

(10) When registering many robots, each is given an initial start location. Sometimes, the location of one robot may overlap with the location of another robot and the two can get stuck together. Adjust the **handleIncomingRequests()** function in the **environmentServer.c** file so that it ensures that each robot is placed a unique non-overlapping location in the environment. That is, make sure that the robot can “move to” (i.e., or be initially placed at) the randomly chosen location. You might want to use a WHILE loop until a good random location is found.

IMPORTANT SUBMISSION INSTRUCTIONS:

Submit all of your **c source code** files as a single **tar** file containing:

1. A **Readme** text file containing
   - your name and studentNumber
   - a list of source files submitted
   - any specific instructions for compiling and/or running your code
2. All of your **.c source** files and all other files needed for testing/running your programs.
3. Any output files required, if there are any.

The code **MUST** compile and run on the course VM.

- If your internet connection at home is down or does not work, we will not accept this as a reason for handing in an assignment late ... so make sure to submit the assignment WELL BEFORE it is due !

- You WILL lose marks on this assignment if any of your files are missing. So, make sure that you hand in the correct files and version of your assignment. You will also lose marks if your code is not **written neatly with proper indentation and containing a reasonable number of comments.** See course notes for examples of what is proper indentation, writing style and reasonable commenting).
