Introduction

I have implemented a simulation of the ringer manager application, following the specifications stated here. The service simulator is described here. The application is coded in `Node.js` and the output can be seen over the command line. I will document the steps to run my program as well as an explanation (a detailed flow can be seen along the comments in my code) of different parts of the same.

Instructions to run my program

1. Install `Node.js` .
2. Run `npm install` at the root directory (`src`) to install any required package (`sync-request`) that my application uses.
3. Run `node main.js` on the command line to execute my application.
4. The program will print out important simulation steps and metrics on the console, as well as all the relationships in the end along with their weights (which have been calculated after accommodating the feedback at every step).
5. The requirement, of course, is that the server at http://yangtze.csc.ncsu.edu:9090/csc555/services.jsp should be up and running.

Workflow of my program

1. It parses relationships from **relationships.txt**. This file contains known relationships (present in the service page) and their corresponding relationship types. The neighbours as well as the callers will affect the decision of the response and the feedback based on their relationship types.
2. Then, it parses known places from **places.txt** which also contains the noise levels shown in the service root page. The noise levels also contribute in the social benefit function.
3. Next, it parses the relationship types from **relationshipTypes.txt** which contains a contribution factor for each relationship type. This will be later used in my social benefit function.
4. It starts simulating visits.
    1. It goes through all places available from the **places.txt** file.
    2. It enters that place through the web service call.
    3. It lists all the neighbors there.
    4. It requests a call in that location.
    5. Calculates the social benefit fucntion based on the neighbors, the call and the place.
    6. Provides the response and gets back a feedback.
    7. Accommodates the feedback into the contribution factors for each relationship.

8. Does the steps `3` to `7` for `5` times in the same place. This is just to run the simulation for more time.
9. Exits the place.

Social benefit function

This function has been completely been implemented along with comments in the `main.js` file. For a detailed workflow, please refer to that file. To give a brief overview of what it does:

1. Iterate over all neighbors.
2. Take the neighbor's weight from a map (this starts as 1, but changes based on the feedback received).
3. If the neighbor is not present in the map from before (this means this neighbor was not listed as a known neighbor in the services page), I insert this neighbor in the map with a relationship type of `stranger`.
4. Multiply the weight with the neighbor relationship type default contribution factor and add to the expected ringer mode property (this starts from 0 for each ringer mode). Thus, family contributes more than stranger to deciding the response type.
5. Add a component to the ringer mode property based on the place as well, depending upon the noise level of the place. If the noise level is from 0 to 2, then the `silent` ringer property gets some points added, 3 to 7 adds to `vibrate` and 8 to 10 adds to `loud`.
6. Find the maximum among the three ringer mode properties. If it is `silent` or `vibrate`, add a component for `vibrate` and `loud` ringer modes respectively based on the urgency of the caller. If the ringer mode chosen is loud, then no need to change it.
7. Find the maximum again and return that response.
8. Basically, in this way every neighbor and caller and place votes with specific weights to the decision of the social ringer manager. And ultimately the highest votes among the three ringer modes wins.

Feedback

Feedback is taken into account by incorporating the same into the weights of the relationships. The following things can happen:

1. If the feedback was positive and the final decision was the same as the expected ringer mode of that user, then nothing is done.
2. If the feedback was positive, but the final decision was not the same, then the weight is reduced a bit, because the user seems unpredictable.
3. If the feedback was negative, but the final decision was the same as the expected ringer mode of that user, then the weight reduced a little more than the last case, because in this case the user is more unpredictable.

4. If the feedback was negative, and the final decision was not the same, then the weight is increased a little bit, so that next time he/she contributes more towards deciding the final response. Basically, giving him a chance to factor into the decision more.
5. If the feedback was neutral, then the weight is reduced a little, because this guy might not have much effect into deciding any response.

Conclusion

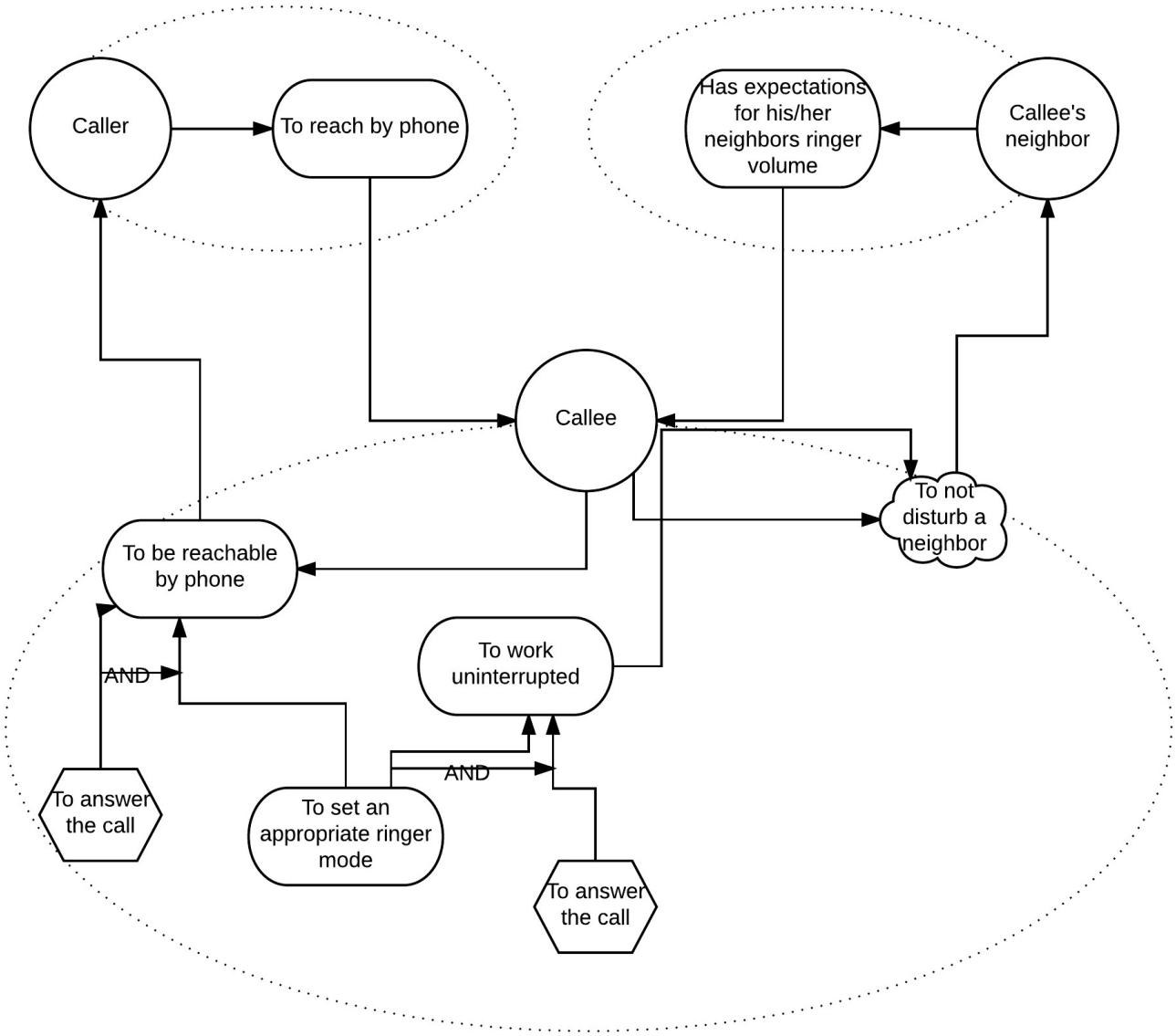
The overall description and workflow of my program and implementation is mentioned in the above parts as well described through the comments in my code. I will also give a snippet of my program output here:

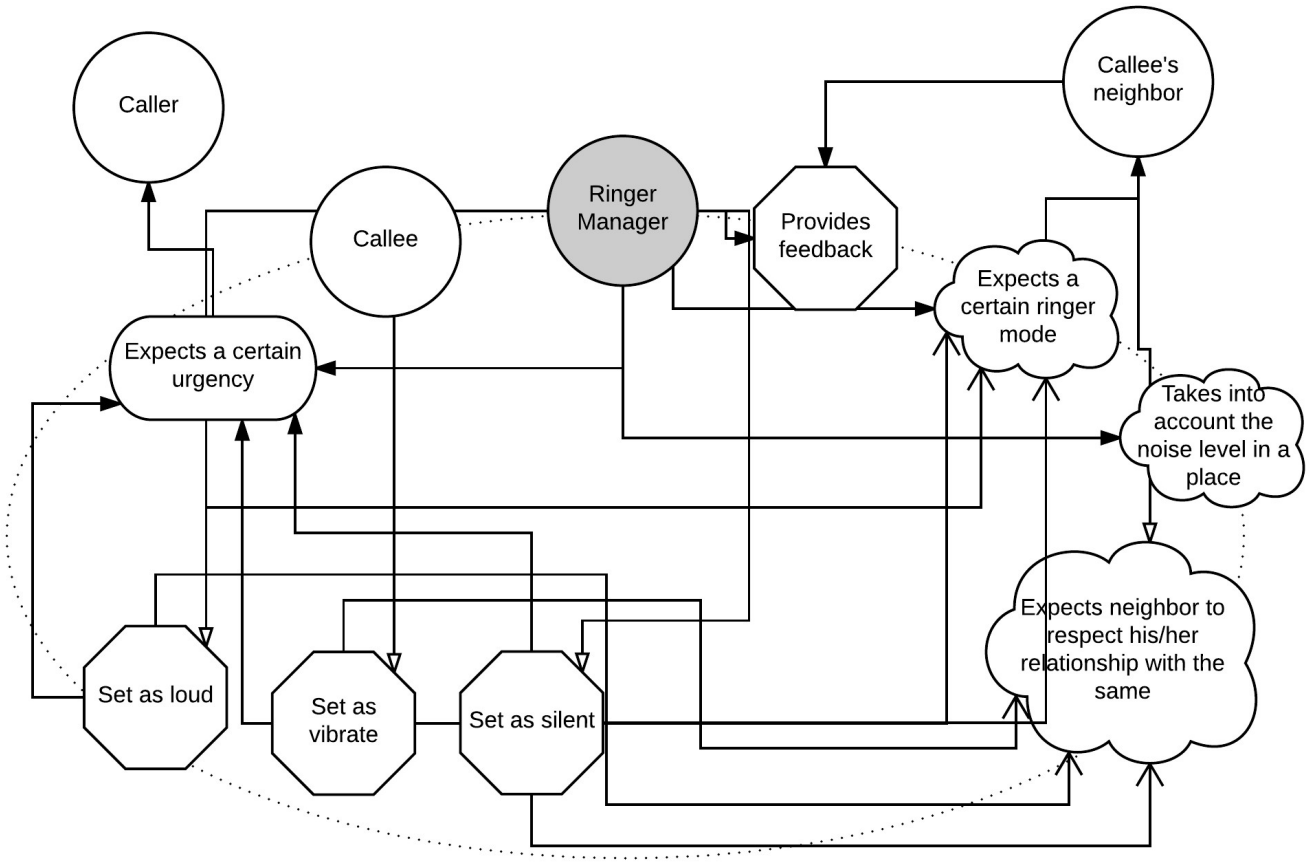
```
Entering hunt.
Getting all neighbors in hunt.
Got call from Denethor.
Options are: {"loud":0.1,"silent":1.7,"vibrate":0.27}
Ringer manager responded with mode silent.
Getting all neighbors in hunt.
Got call from Denethor.
Options are: {"loud":0.097,"silent":1.649,"vibrate":0.21000000000000002}
Ringer manager responded with mode silent.
Getting all neighbors in hunt.
Got call from Radagast.
Options are: {"loud":0.10700000000000001,"silent":1.601,"vibrate":0.22}
Ringer manager responded with mode silent.
Getting all neighbors in hunt.
Got call from Nazgul.
Options are: {"loud":0.11700000000000002,"silent":1.571,"vibrate":0.19}
Ringer manager responded with mode silent.
Getting all neighbors in hunt.
Got call from Bilbo.
Options are: {"loud":0.11400000000000002,"silent":1.517,"vibrate":0.2999999999
9999993}
Ringer manager responded with mode silent.
Exiting hunt.

...

Relationships at the end of iteration:
{ Arwen: { relationship: 'family', weight: 0.8899999999999999 },
  Bilbo: { relationship: 'friend', weight: 0.97 },
  Ceorl: { relationship: 'colleague', weight: 0.47999999999999987 },
  Denethor: { relationship: 'stranger', weight: 0.75 },
  Elrond: { relationship: 'family', weight: 0.7099999999999999 },
  Faramir: { relationship: 'friend', weight: 0.7199999999999998 },
  Gandalf: { relationship: 'colleague', weight: 0.95 },
```

```
  Hurin: { relationship: 'stranger', weight: 1 },
  Isildur: { relationship: 'family', weight: 0.9199999999999998 },
  Legolas: { relationship: 'friend', weight: 0.5000000000000001 },
  Maggot: { relationship: 'colleague', weight: 0.8499999999999999 },
  Nazgul: { relationship: 'stranger', weight: 0.8899999999999999 },
  Orophin: { relationship: 'family', weight: 0.94 },
  Radagast: { relationship: 'friend', weight: 0.8599999999999999 },
  Sauron: { relationship: 'colleague', weight: 0.94 },
  Robin: { relationship: 'stranger', weight: 0.9099999999999999 },
  'Nick Fury': { relationship: 'stranger', weight: 0.7499999999999998 },
  'Wonder Woman': { relationship: 'stranger', weight: 0.9099999999999999 },
  Beast: { relationship: 'stranger', weight: 1.2400000000000002 },
  'Martian Manhunter': { relationship: 'stranger', weight: 0.879999999999999
9 },
  Nightcrawler: { relationship: 'stranger', weight: 0.97 },
  'Barry Allen (The Flash)': { relationship: 'stranger', weight: 0.74999999999
99998 },
  'Black Panther': { relationship: 'stranger', weight: 0.8799999999999999 },
  'Green Arrow': { relationship: 'stranger', weight: 0.9099999999999999 },
  'Captain America': { relationship: 'stranger', weight: 1.5000000000000004 },
  'James Gordon': { relationship: 'stranger', weight: 0.8499999999999999 },
  'Ant-Man': { relationship: 'stranger', weight: 1.5000000000000004 },
  Thor: { relationship: 'stranger', weight: 1.5000000000000004 },
  Aquaman: { relationship: 'stranger', weight: 1.3700000000000003 },
  'Invisible Woman': { relationship: 'stranger', weight: 0.8499999999999999 },
  Daredevil: { relationship: 'stranger', weight: 0.94 },
  Wolverine: { relationship: 'stranger', weight: 1.5000000000000004 } }
```

As you can see, it entered `Hunt` and then got the list of neighbours, did a loop of requesting 5 calls and responding to them. At the end I have also shown the status of relationships and how much their weights are, which denotes how much their expected ringer mode should contribute towards deciding the final ringer mode. So the ones which are insanely grumpy, have their weights probably reduced quite a lot, and they don't affect the outcome of the decision of the ringer manager as much. Another thing to note is the fact that this list does not only contain the previous relationships that I knew about, but also random ones (rather the developers/students themselves who have entered the location) which the service have provided me with. To me, they are strangers and their feedback are automatically infused into the decision making process.

This forms my final project report. Below I have also included the `system-as-is` and `system-to-be` models, but they basically have remained the same as the last submission. Only the `system-to-be` model has now incorporated the noise levels in places.


Caller
To reach by phone
Has expectations for his/her neighbors ringer volume
Callee's neighbor
Callee
To not disturb a neighbor
To be reachable by phone
To work uninterrupted
AND
AND
To answer the call
To set an appropriate ringer mode
To answer the call


Caller
Callee's neighbor
Callee
Ringer Manager
Provides feedback
Expects a certain ringer mode
Expects a certain urgency
Takes into account the noise level in a place
Set as loud
Set as vibrate
Set as silent
Expects neighbor to respect his/her relationship with the same