



Message Broadcast in Dynamic Networks

Simulation with Repast Symphony

Antonio Bucchiarone

Fondazione Bruno Kessler, Trento – Italy

bucchiarone@fbk.eu

14 October 2019

- Agent-based model involving *zombies chasing humans* and *humans running away from zombies*.
- *Zombie* and *Human* classes.
- **Zombie Behavior:**
 - To wander around *looking for Humans to infect*.
 - Each iteration of the simulation, each *Zombie will determine where the most Humans are* within its local area and move there.
 - Once *there it will attempt to infect a Human* at that location and *turn it into a Zombie*.

- Zombies and Humans are located within a **ContinuousSpace** and a **Grid**.
- A **ContinuousSpace** allows us to use floating point numbers (e.g., 1.5) as the coordinates of a Zombie's and Human's location.
- The **Grid** allow us to do neighbourhood and proximity queries (i.e., "*who is near me?*") using discrete integer Grid coordinates.

```
1 public class Zombie {  
2  
3     private ContinuousSpace<Object> space;  
4     private Grid<Object> grid;  
5  
6     public Zombie(ContinuousSpace<Object> space, Grid<Object> grid) {  
7         this.space = space;  
8         this.grid = grid;  
9     }  
10 }
```

- Space and grid in which the zombie will be located.
- The constructor which sets the values of the **space** and the **grid** variables.

```
1 public void step() {
2     // get the grid location of this Zombie
3     GridPoint pt = grid.getLocation(this);
4
5     // use the GridCellNgh class to create GridCells for
6     // the surrounding neighborhood.
7     GridCellNgh<Human> nghCreator = new GridCellNgh<Human>(grid, pt,
8         Human.class, 1, 1);
9     // import repast.simphony.query.space.grid.GridCell
10    List<GridCell<Human>> gridCells = nghCreator.getNeighborhood(true)
11    SimUtilities.shuffle(gridCells, RandomHelper.getUniform());
12
13    GridPoint pointWithMostHumans = null;
14    int maxCount = -1;
15    for (GridCell<Human> cell : gridCells) {
16        if (cell.size() > maxCount) {
17            pointWithMostHumans = cell.getPoint();
18            maxCount = cell.size();
19        }
20    }
21 }
```

- It will be called every iteration of the simulation.
- A *GridCellNgh* is used to create a List of *GridCells* containing Humans.
- **GridCell**'s size is a measure of the number of object it contains (i.e., the greatest size contains the most Humans).

- To move agents (i.e., zombies) towards a specific location
 - the discovered location with the most Humans.

```
1 public void moveTowards(GridPoint pt) {
2     // only move if we are not already in this grid location
3     if (!pt.equals(grid.getLocation(this))) {
4         NdPoint myPoint = space.getLocation(this);
5         NdPoint otherPoint = new NdPoint(pt.getX(), pt.getY());
6         double angle = SpatialMath.calcAngleFor2DMovement(space,
7             myPoint, otherPoint);
8         space.moveByVector(this, 1, angle, 0);
9         myPoint = space.getLocation(this);
10        grid.moveTo(this, (int)myPoint.getX(), (int)myPoint.getY());
11    }
12 }
```

- **NdPoint** stores the agent coordinates as doubles.
- **SpatialMath** method **calcAngleFor2DMovement** is used to calculate the *angle* along which agents should move if they *move towards the GridPoint*.

```
1 for (GridCell<Human> cell : gridCells) {
2     if (cell.size() > maxCount) {
3         pointWithMostHumans = cell.getPoint();
4         maxCount = cell.size();
5     }
6 }
7 moveTowards(pointWithMostHumans);
```

```
1 @ScheduledMethod(start = 1, interval = 1)
2 public void step() {
3     // get the grid location of this Zombie
4     GridPoint pt = grid.getLocation(this);
```

- The *step* method is called every iteration of the simulation.

```
1 public class Human {
2
3     private ContinuousSpace<Object> space;
4     private Grid<Object> grid;
5     private int energy, startingEnergy;
6     public Human(ContinuousSpace<Object> space, Grid<Object> grid,
7         int energy)
8     {
9         this.space = space;
10        this.grid = grid;
11        this.energy = startingEnergy = energy;
12    }
13    ...
14 }
```

- It will be used to track the *current amount of energy* a Human has.

Agent Method – run()

```
1 public void run() {
2     // get the grid location of this Human
3     GridPoint pt = grid.getLocation(this);
4     // use the GridCellNgh class to create GridCells for
5     // the surrounding neighborhood.
6     GridCellNgh<Zombie> nghCreator = new GridCellNgh<Zombie>(grid, pt,
7         Zombie.class, 1, 1);
8     List<GridCell<Zombie>> gridCells = nghCreator.getNeighborhood(true);
9     SimUtilities.shuffle(gridCells, RandomHelper.getUniform());
10
11     GridPoint pointWithLeastZombies = null;
12     int minCount = Integer.MAX_VALUE;
13     for (GridCell<Zombie> cell : gridCells) {
14         if (cell.size() < minCount) {
15             pointWithLeastZombies = cell.getPoint();
16             minCount = cell.size();
17         }
18     }
19
20     if (energy > 0) {
21         moveTowards(pointWithLeastZombies);
22     } else {
23         energy = startingEnergy;
24     }
25 }
```

- It determines which cells has the least Zombies and attempts to move towards that.
- *moveTowards* is only called if the energy level is greater than 0

Agent Method – moveTowards()

```
1 public void moveTowards(GridPoint pt) {
2     // only move if we are not already in this grid location
3     if (!pt.equals(grid.getLocation(this))) {
4         NdPoint myPoint = space.getLocation(this);
5         NdPoint otherPoint = new NdPoint(pt.getX(), pt.getY());
6         double angle = SpatialMath.calcAngleFor2DMovement(space, myPoint,
7             otherPoint);
8         space.moveByVector(this, 2, angle, 0);
9         myPoint = space.getLocation(this);
10        grid.moveTo(this, (int)myPoint.getX(), (int)myPoint.getY());
11        energy--;
12    }
13 }
```

- The energy is decremented and the human moves 2 units rather than 1.

```
1 @Watch(watcheeClassName = "jzombies.Zombie",
2       watcheeFieldNames = "moved",
3       query = "within_moore 1",
4       whenToTrigger = WatcherTriggerSchedule.IMMEDIATE)
5 public void run() {
```

- A *watcher* will trigger the `run()` method whenever a `Zombie` moves into a `Human`'s neighbourhood.
- This `Watch` will watch for any changes to a `"moved"` variable in the `Zombies` class.
- Whenever any `Zombie` moves and their `moved` variable is updated, then the *Watch* will be checked for each `Human`.
- *If the query returns true* for that particular `Human` then *run* will be called immediately on that `Human`.
- The query returns **true** when the `Zombie` that moved is within the Moore neighbourhood (8 surrounding grid cells) of the `Human` whose `Watch` is currently being evaluated.

```
1 @Watch(watcheeClassName = "jzombies.Zombie",
2       watcheeFieldNames = "moved",
3       query = "within_moore 1",
4       whenToTrigger = WatcherTriggerSchedule.IMMEDIATE)
5 public void run() {
```

- A *watcher* will trigger the `run()` method whenever a `Zombie` moves into a `Human`'s neighbourhood.
- This `Watch` will watch for any changes to a `"moved"` variable in the `Zombies` class.
- Whenever any `Zombie` moves and their `moved` variable is updated, then the *Watch* will be checked for each `Human`.
- If the query returns *true* for that particular `Human` then *run* will be called immediately on that `Human`.
- The query returns **true** when the `Zombie` that moved is within the Moore neighbourhood (8 surrounding grid cells) of the `Human` whose `Watch` is currently being evaluated.

- **ContextBuilder** Class used to build the *Simulation Context*.
- A *Context* is essentially a named set of agents.
- **Projections**: it takes the *agents in a Context* and imposes some sort of *structure* on them.
 - **ContinuousSpace** and **Grid** are projections.
 - They take agents and locate them in a continuous space and matrix like grid respectively.

```
1 public Context build(Context<Object> context) {
2     context.setId("jzombies");
3
4     ContinuousSpaceFactory spaceFactory =
5         ContinuousSpaceFactoryFinder.createContinuousSpaceFactory(null);
6     ContinuousSpace<Object> space =
7         spaceFactory.createContinuousSpace("space", context,
8             new RandomCartesianAdder<Object>(),
9             new repast.simphony.space.continuous.WrapAroundBorders(),
10            50, 50);
11
12     GridFactory gridFactory = GridFactoryFinder.createGridFactory(null);
13     // Correct import: import repast.simphony.space.grid.WrapAroundBorders;
14     Grid<Object> grid = gridFactory.createGrid("grid", context,
15         new GridBuilderParameters<Object>(new WrapAroundBorders(),
16             new SimpleGridAdder<Object>(),
17             true, 50, 50));
18
19     return context;
20 }
```

```
1 int zombieCount = 5;
2 for (int i = 0; i < zombieCount; i++) {
3     context.add(new Zombie(space, grid));
4 }
5
6 int humanCount = 100;
7 for (int i = 0; i < humanCount; i++) {
8     int energy = RandomHelper.nextIntFromTo(4, 10);
9     context.add(new Human(space, grid, energy));
10 }
```

- Agents are added to the space and grid using their **Adders**.
- The Humans are created with a *random energy level* from 4 to 10.

```
for (Object obj : context) {
    NdPoint pt = space.getLocation(obj);
    grid.moveTo(obj, (int)pt.getX(), (int)pt.getY());
}
```

- To move the agents to the **Grid** location that corresponds to their **ContinuousSpace** location.

```
1 public void infect() {
2     GridPoint pt = grid.getLocation(this);
3     List<Object> humans = new ArrayList<Object>();
4     for (Object obj : grid.getObjectsAt(pt.getX(), pt.getY())) {
5         if (obj instanceof Human) {
6             humans.add(obj);
7         }
8     }
9     if (humans.size() > 0) {
10        int index = RandomHelper.nextIntFromTo(0, humans.size() - 1);
11        Object obj = humans.get(index);
12        NdPoint spacePt = space.getLocation(obj);
13        Context<Object> context = ContextUtils.getContext(obj);
14        context.remove(obj);
15        Zombie zombie = new Zombie(space, grid);
16        context.add(zombie);
17        space.moveTo(zombie, spacePt.getX(), spacePt.getY());
18        grid.moveTo(zombie, pt.getX(), pt.getY());
19
20        Network<Object> net = (Network<Object>)context.
21            getProjection("infection network");
22        net.addEdge(this, zombie);
23    }
24 }
```

- A Human is chosen at random from the Humans at *the Zombie's grid location*.
- The chosen *Human is removed from the simulation and replaced with a Zombie*.

- Data Collection and Visualization in Repast Symphony.
- First assignment Description
 - **Deadline 1:** Friday 22, November - 6pm
 - **Deadline 2:** Friday 20, December – 6pm
 - **Deadline 3:** Thursday 16, January – 6pm

<https://bucchiarone.bitbucket.io/>



Message Broadcast in Dynamic Networks

Simulation with Repast Symphony

Antonio Bucchiarone

Fondazione Bruno Kessler, Trento – Italy

bucchiarone@fbk.eu

14 October 2019

- To simulate message passing in highly mobile multi-hop ad-hoc networks.
- Broadcasting of messages by treating the nodes in the *network as autonomous agents*.
- Set of rules governing the passing of messages from one node to another.
- The portions of the network at any given time can change.
- A baseline is established: agents in the system can move after each transmittal of a message to their immediate neighbours.

https://www3.nd.edu/~agent/Papers/C028_Kirkpatrick_Madey_final.pdf

- All the nodes of the network are *mobile*.
 - **Ex1:** Robots exploring the surface of a distant planet.
 - **Ex2:** Soldiers moving on a battlefield.
 - **Ex3:** Robots searching for available victims.
 - **Ex4:** The elimination of fixed towers in cellular telephone systems.
- Mobile independently operating agents that seemingly *work together to perform tasks*:
 - without complex communication networks;
 - without global knowledge of the locations of individuals.
- The flock appears to be moving in an intelligent, directed manner

Mitchel Resnick: **Turtles, termites, and traffic jams - explorations in massively parallel microworlds**. MIT Press 1998, ISBN 978-0-262-68093-6, pp. I-XVIII, 1-163

- Group performing tasks effectively by using only a small set of rules for individual behavior.
- The topology of such networks is generally fixed
 - Changes may result from equipment failures or addition/removal of nodes.



- Swarm Intelligence to networks that have frequent topology changes.
- Simulation of the networks and using the simulations to study message passing within them.
- We want to explore the potential for providing efficient and reliable communication between nodes in a mobile network.

- Can communicate with other agents;
- Can sense and react to changes in the environment;
- Are capable of long periods of unattended operation;
- There is no central authority governing an agent's behavior.

- The environment is simulated as a rectangular/square grid of cells.
- A cell is either empty or occupied by an agent.
- The grid is populated by placing agents in some given percentage of the total number of grid locations.
- The percentage is called the *density* of agents in the grid.
- The *dimensions of the grid* and the *density* of agents are selected at the beginning of each run.

- An Agent is selected to be *the originator of the message*.
- In some runs a *destination agent* is selected as well.
- In others *the message is sent to all agents*.
- As the simulation runs, individual agents move randomly, in any of eight possible directions (N, NE, E, etc...).
- Only one agent can occupy a given grid location at a time
 - If moving in a selected direction would result in a collision, the agent simply sits still for one time step and attempts to move again (in a randomly selected direction) at the next step.

- The simulations run can be categorized into four types.
 - **Case 1:** Messages are passed only to neighbors (that is, agents in any of the eight locations surrounding the agent carrying the message) at each step.
 - **Case 2:** Messages are passed throughout the ad hoc networks at each step.
 - **Case 3:** Messages are passed as in Case 1 but some agents move in a fixed direction, reversing at the edges.
 - **Case 4:** Messages are passed as in Case 1 but are purged from the agent's memory after some period of time and an agent, once having carried the message, will not accept it a second time.
- Except for the fixed direction agents in Case 3, the agents move randomly at each step and pass messages to other agents with which they have contact – either direct or networked.