Message Broadcast in Dynamic Networks
Simulation with Repast Simphony

Antonio Bucchiarone

Fondazione Bruno Kessler, Trento – Italy
bucchiarone@fbk.eu

07 October 2020

Repast Java Through an Example

- 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*.

Agents Movement

- 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.

Agent field variables and constructor

```java
public class Zombie {

    private ContinuousSpace<Object> space;
    private Grid<Object> grid;

    public Zombie(ContinuousSpace<Object> space, Grid<Object> grid) {
        this.space = space;
        this.grid = grid;
    }
}
```

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

Agent Method – Step()


```java
public void step() {
   // get the grid location of this Zombie
   GridPoint pt = grid.getLocation(this);

   // use the GridCellNgh class to create GridCells for
   // the surrounding neighborhood.
   GridCellNgh<Human> nghCreator = new GridCellNgh<Human>(grid, pt,
      Human.class, 1, 1);
   // import repast.simphony.query.space.grid.GridCell
   List<GridCell<Human>> gridCells = nghCreator.getNeighborhood(true)
   SimUtilities.shuffle(gridCells, RandomHelper.getUniform());

   GridPoint pointWithMostHumans = null;
   int maxCount = -1;
   for (GridCell<Human> cell : gridCells) {
      if (cell.size() > maxCount) {
         pointWithMostHumans = cell.getPoint();
         maxCount = cell.size();
      }
   }
}
```

- 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).

Agent Method – moveTowards()

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

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

- **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.*

Agent Method Annotation

```
for (GridCell<Human> cell : gridCells) {
    if (cell.size() > maxCount) {
        pointWithMostHumans = cell.getPoint();
        maxCount = cell.size();
    }
}
moveTowards(pointWithMostHumans);
```

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

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

Another Agent - Human

```java
public class Human {

    private ContinuousSpace<Object> space;
    private Grid<Object> grid;
    private int energy, startingEnergy;
    public Human(ContinuousSpace<Object> space, Grid<Object> grid,
        int energy)
    {
        this.space = space;
        this.grid = grid;
        this.energy = startingEnergy = energy;
    }
...
}
```

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

Agent Method – run()

```java
public void run() {
   // get the grid location of this Human
   GridPoint pt = grid.getLocation(this);
   // use the GridCellNgh class to create GridCells for
   // the surrounding neighborhood.
   GridCellNgh<Zombie> nghCreator = new GridCellNgh<Zombie>(grid, pt,
         Zombie.class, 1, 1);
   List<GridCell<Zombie>> gridCells = nghCreator.getNeighborhood(true);
   SimUtilities.shuffle(gridCells, RandomHelper.getUniform());

   GridPoint pointWithLeastZombies = null;
   int minCount = Integer.MAX_VALUE;
   for (GridCell<Zombie> cell : gridCells) {
      if (cell.size() < minCount) {
         pointWithLeastZombies = cell.getPoint();
         minCount = cell.size();
      }
   }

   if (energy > 0) {
      moveTowards(pointWithLeastZombies);
   } else {
      energy = startingEnergy;
   }
}
```

- 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()

```java
public void moveTowards(GridPoint pt) {
    // only move if we are not already in this grid location
    if (!pt.equals(grid.getLocation(this))) {
        NdPoint myPoint = space.getLocation(this);
        NdPoint otherPoint = new NdPoint(pt.getX(), pt.getY());
        double angle = SpatialMath.calcAngleFor2DMovement(space, myPoint,
            otherPoint);
        space.moveByVector(this, 2, angle, 0);
        myPoint = space.getLocation(this);
        grid.moveTo(this, (int)myPoint.getX(), (int)myPoint.getY());
        energy--;
    }
}
```

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

@Watch Annotation

```
@Watch(watcheeClassName = "jzombies.Zombie",
   watcheeFieldNames = "moved",
   query = "within_moore 1",
   whenToTrigger = WatcherTriggerSchedule.IMMEDIATE)
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.

@Watch Annotation

```
@Watch(watcheeClassName = "jzombies.Zombie",
   watcheeFieldNames = "moved",
   query = "within_moore 1",
   whenToTrigger = WatcherTriggerSchedule.IMMEDIATE)
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.

Simulation Initialization

- **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.

```
public Context build(Context<Object> context) {
   context.setId("jzombies");

   ContinuousSpaceFactory spaceFactory =
      ContinuousSpaceFactoryFinder.createContinuousSpaceFactory(null);
   ContinuousSpace<Object> space =
      spaceFactory.createContinuousSpace("space", context,
         new RandomCartesianAdder<Object>(),
         new repast.simphony.space.continuous.WrapAroundBorders(),
         50, 50);

   GridFactory gridFactory = GridFactoryFinder.createGridFactory(null);
   // Correct import:  import repast.simphony.space.grid.WrapAroundBorders;
   Grid<Object> grid = gridFactory.createGrid("grid", context,
           new GridBuilderParameters<Object>(new WrapAroundBorders(),
           new SimpleGridAdder<Object>(),
            true, 50, 50));

   return context;
}
```

Add Agents to the Context

```
int zombieCount = 5;
for (int i = 0; i < zombieCount; i++) {
    context.add(new Zombie(space, grid));
}

int humanCount = 100;
for (int i = 0; i < humanCount; i++) {
    int energy = RandomHelper.nextIntFromTo(4, 10);
    context.add(new Human(space, grid, energy));
}
```

- 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.

Infect() Method

```
public void infect() {
   GridPoint pt = grid.getLocation(this);
   List<Object> humans = new ArrayList<Object>();
   for (Object obj : grid.getObjectsAt(pt.getX(), pt.getY())) {
      if (obj instanceof Human) {
         humans.add(obj);
      }
   }
   if (humans.size() > 0) {
      int index = RandomHelper.nextIntFromTo(0, humans.size() - 1);
      Object obj = humans.get(index);
      NdPoint spacePt = space.getLocation(obj);
      Context<Object> context = ContextUtils.getContext(obj);
      context.remove(obj);
      Zombie zombie = new Zombie(space, grid);
      context.add(zombie);
      space.moveTo(zombie, spacePt.getX(), spacePt.getY());
      grid.moveTo(zombie, pt.getX(), pt.getY());

      Network<Object> net = (Network<Object>)context.
         getProjection("infection network");
      net.addEdge(this, zombie);
   }
}
```

- 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.*

Next Lecture

- Data Collection and Visualization in Repast Simphony.

- First assignment Description

  - **Deadline 1:** Friday 22, November - 6pm
  - **Deadline 2**: Friday 20, December – 6pm

https://bucchiarone.bitbucket.io/

Message Broadcast in Dynamic Networks
Simulation with Repast Simphony

Antonio Bucchiarone

Fondazione Bruno Kessler, Trento – Italy
bucchiarone@fbk.eu

14 October 2019

Introduction

- 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

Mobile Network

- 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

Swarm Intelligence

- 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 of 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.

Agents Behavior

- 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 Context

- 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.

Simulation Behavior

FONDAZIONE BRUNO KESSLER

- 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.

Simulation Cases

- 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.