MAAS - Project Report
Commitment Issues

Sushant Vijay Chavan
Ahmed Faisal Abdelrahman
Abanoub Abdelmalak

January 19, 2019

Contents

1 Introduction **3**

2 Packaging Stage **3**
- 2.1 Agents . . . 4
  - 2.1.1 Entry Agent (Developer: Sushant) . . . 4
  - 2.1.2 Generic Item Processor (Developer: Abanoub) . . . 5
  - 2.1.3 Packaging Agent (Developer: Sushant) . . . 5
  - 2.1.4 Loading Bay Agent (Developer: Ahmed) . . . 5

3 Delivery Stage **7**
- 3.1 Agents . . . 7
  - 3.1.1 Order Aggregator (Developer: Abanoub) . . . 7
  - 3.1.2 Transport Agent (Developer: Abanoub) . . . 8
  - 3.1.3 Truck Agent (Developer: Sushant) . . . 8
  - 3.1.4 Street Network Agent (Developer: Ahmed) . . . 8
  - 3.1.5 Mailbox Agent (Developer: Ahmed) . . . 9
- 3.2 Objects . . . 10
  - 3.2.1 Graph . . . 10
  - 3.2.2 Vertex . . . 10
  - 3.2.3 Edge . . . 10
  - 3.2.4 DijkstraAlgorithm . . . 10
  - 3.2.5 Box . . . 10
- 3.3 Graph Visualization Agent (Developer: All) . . . 11

4 Usage Instructions **12**
- 4.1 Running the two stages . . . 12
- 4.2 Running on distributed systems . . . 12

1 Introduction

In this project, we implemented two stages and a visualization component of the Flying Saucers Bakery, an inter-group cooperative project for the Multi-Agent and Agent Systems course. The stages this group was responsible for were the final two: packaging and delivery. Initially, the group was tasked with formulating an architecture for the entire project, including specifying the agents, their communication mechanisms, and the flow of a bakery's processes. Subsequently, the group assumed responsibility for the afore-mentioned stages, implementing them end-to-end, and working to integrate them with the agents developed by other teams.

As per the specifications of the project, the packaging stage was developed so as to receive baked bakery products, perform certain post-baking steps, and packaging them into separate boxes to be passed on to the next stage. The delivery stage was designed to receive these, aggregate them based on customer orders, and facilitate delivering them to the right customers most efficiently. This involved distributing them among trucks, which travel around the street network to deliver orders from bakeries to customers. A graph visualizer application of the street network was developed to enable visualization of this stage.

All agents were defined in each stage to be responsible for a certain task, and the design decisions were agreed upon among the group members. Each member was then responsible for developing certain agents, while adhering to the defined architecture and agent communication constraints. In addition, certain data structures were encapsulated in objects. These were needed for certain functionalities, but did not justify implementation as agents, since no communication or decision-making was made on their part.

Throughout the project, developed agents had to be appropriately maintained and updated to keep up with changes to the upstream repository of the project. This was particularly necessary for the interface agents (loading bay and mailbox agents), since they were standardized and used by multiple groups. During the project git issues, pull requests, reviews, and other tools were utilized to enable seamless integration with the other groups.

This report is structured as follows. Section 2 and 3 elaborate on the packaging and delivery stages, respectively, including descriptions of the architectures, the developed agents, and message specifications for the interface agents. Section 3.2 explains the graph visualization of the delivery phase's street network. Finally, section 4 contains instructions for running both stages, and running separate stages on separate machines.

2 Packaging Stage

This stage deals with completion of the post baking stages and then packaging the products into boxes so that they are ready for delivery. The stage receives the baked and cooled products from the cooling racks and preforms the final preparation of the products such as sprinkling and decorating according to the recipe for the product. Once the products are ready, they are packed into boxes based on the delivery date priority of the orders. Once all the boxes of a given product type in an order are ready, they are sent to the delivery stage for transportation.

Figure 1 gives an overview of the architecture of the packaging stage. This stage consists of three agents and in addition communicates with the CoolingRacks agent to receive the cooled

products. The GenericItemProcessor takes care of all the post-baking steps, where as the packaging agent boxes the products based on the order priority. The Loading bay keeps track of the produced quantity of the products and informs the delivery stage whenever all products of a type in an order are ready.

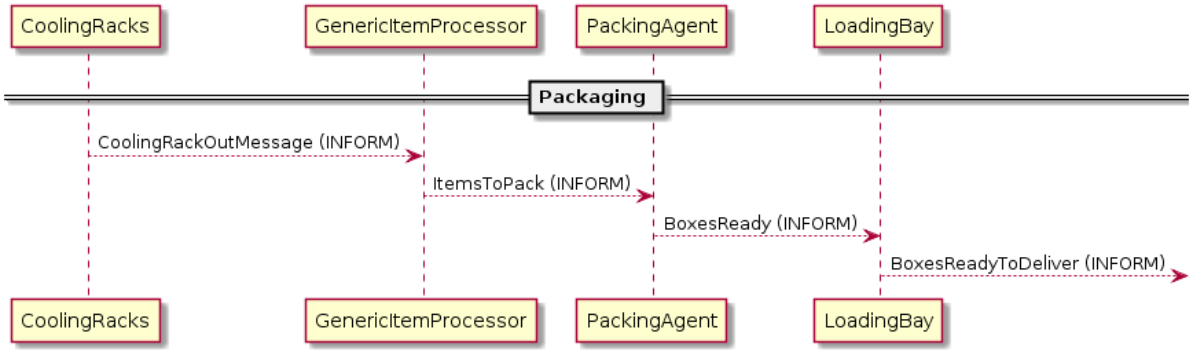

Figure 1: Packaging Stage Architecture

The output of this stage is a set of boxes that consists of a fulfilled item from a customer order. For example, if a customer order consists of 10 breads and 15 donuts, assuming that the loading bay sees that the 15 donuts of this order are available, it sends these boxes to the delivery stage. Please note that not all products of this order are yet complete and that the loading bay will wait until all the breads of this order are available before sending out a new message to the delivery stage.

2.1 Agents

2.1.1 Entry Agent (Developer: Sushant)

In order to test the two developed stages independently of the other stages, we need an agent that can trigger the cooling racks agent to start the simulation. The EntryAgent fulfills this requirement. Using this agent, it is possible to simulate that the products from of an order have been baked and are ready for cooling, which then starts the two following stages until the order has been delivered to the customer. It should be noted that the EntryAgent simply uses the order information available in the scenario files and does not generate any data on its own.

The Entry agent has two functionalities. First, it reads the order details from the currently active scenario directory and then generates dummy messages that inform the cooling rack that the products of an order are ready. These messages are then sent to the cooling rack at the delivery time mentioned in the scenario file. Therefore, the first part of this agent acts as the trigger for the two stages. Since the order processor agent was also not available on the upstream repository, we added the order processor functionality as the second functionality of the entry agent. Whenever an order is sent to the cooling rack, a corresponding order information is also broadcast to all agents. We used the same message format as the actual order processor. In this way we ensure that all our agents are compatible with the actual order processor, whenever it is available.

2.1.2 Generic Item Processor (Developer: Abanoub)

This agent is responsible for receiving the items sent by the cooling racks, prepare them for packaging and send them to the packaging agent to be packed. Based on the scenario files used for the simulation and the bakery this agent belongs to, it creates a list with all the products. Using the scenario file, each product is acquired with the processes it needs to be ready for packaging and their duration. After preparing this list of the products with their features, the agent starts to receive products from the cooling racks and start executing the processes one by one for each of them until it is done. The products are sent to the packaging agent as soon as they are processed and all the time needed for each of the processes is finished.

2.1.3 Packaging Agent (Developer: Sushant)

This agent is responsible for receiving the ready products and then packaging them into boxes. The packaged boxes are then sent to the loading bay so that they can be dispatched to the customer. This agent prioritizes the packaging of the received products based on the delivery time of the orders.

For example, assume we have two orders requiring 10 and 20 breads respectively and that the delivery date of the first order is earlier than that of the second. Now if we receive 20 breads from the previous stage, the packaging agents first boxes the 10 breads for the first order and sends them to the loading bay. It then packages the remaining 10 breads as a part of the second order. It must be noted that the packaging agent sends out the boxes as soon as they are full. It expects the loading bay to store the boxes until all boxes of a product type in an order are ready.

2.1.4 Loading Bay Agent (Developer: Ahmed)

The loading bay agent acts as the interface between the packaging stage and the delivery stage. Aside from transferring boxes of packaged products along from the packaging agent to the order aggregator agent, it is responsible for sending boxes of products only when all of the products of a certain type, in a respective order have been fulfilled.

The agent receives two types of messages, one from the preceding packaging agent and another from the common order processing agent. The former sends a list of boxes of products that belong to an order, while the latter provides the list of products needed to fulfill every order, among the other order details. The agent receives boxes of products and retains them with their order IDs. Meanwhile, it maintains all orders' details and continuously checks if it has all the products of a certain type needed for any order. Once an order's products of a particular type are all present, it dispatches a message to the next agent containing a list of all the boxes containing these products.

Message Descriptions:

(Incoming) Packaged Boxes Message:

Description: The LoadingBayAgent receives boxes of packaged products from a packaging agent in a message of the following format:
Performative: INFORM
Sender: AID of the packaging agent
Receiver: AID of the LoadingBayAgent
Conversation ID: "boxes-ready"

Figure 2 shows an example of the contents of the message.

```
{
  "OrderID": "order-001",
  "Boxes": [
    {
      "BoxID": "1",
      "ProductType": "Bread",
      "Quantity": 12
    },
    {
      "BoxID": "2",
      "ProductType": "Bread",
      "Quantity": 6
    },
    {
      "BoxID": "3",
      "ProductType": "Donut",
      "Quantity": 5
    }
  ]
}
```

Figure 2: Packaged Boxes Example Message

(Outgoing) Fulfilled Order Boxes Message:

Description: The LoadingBayAgent agent sends the order aggregator agent a message containing the details of boxes of products belonging to a particular order. It does so once it receives boxes that contain all of the products of a type that fulfill that order. The message is of the following format:
Performative: INFORM
Sender: AID of the LoadingBayAgent
Receiver: AID of an OrderAggregatorAgent

The message is of the same format as the incoming message, and an example can be seen on Figure 2.

3 Delivery Stage

This stage handles the delivery of the packaged boxes from the bakery to the customers. The inputs to this stage are a set of boxes from the packaging stage as described in the section 2. This stage waits until all the products of an order are ready for dispatch and then requests the trucks for transportation of these boxes. Once the orders have been delivered to the customer, the trucks post the status of the delivery to the mailbox, which then informs the concerned customer agent.

This stage consists of five agents as shown in the architecture diagram in figure 3. In addition, this stage talks with two other agents: *LoadingBayAgent* and the *GraphVisualizationAgent*. The figure gives an overview of the communication happening between the agents in order to deliver an order received from the LoadingBayAgent. The visualization of the delivery stage will be explained in detail in section 3.3. However, it is helpful to know that the only agents that need to communicate with the GraphVisualizationAgent are the StreetNetworkAgent and the TruckAgents.

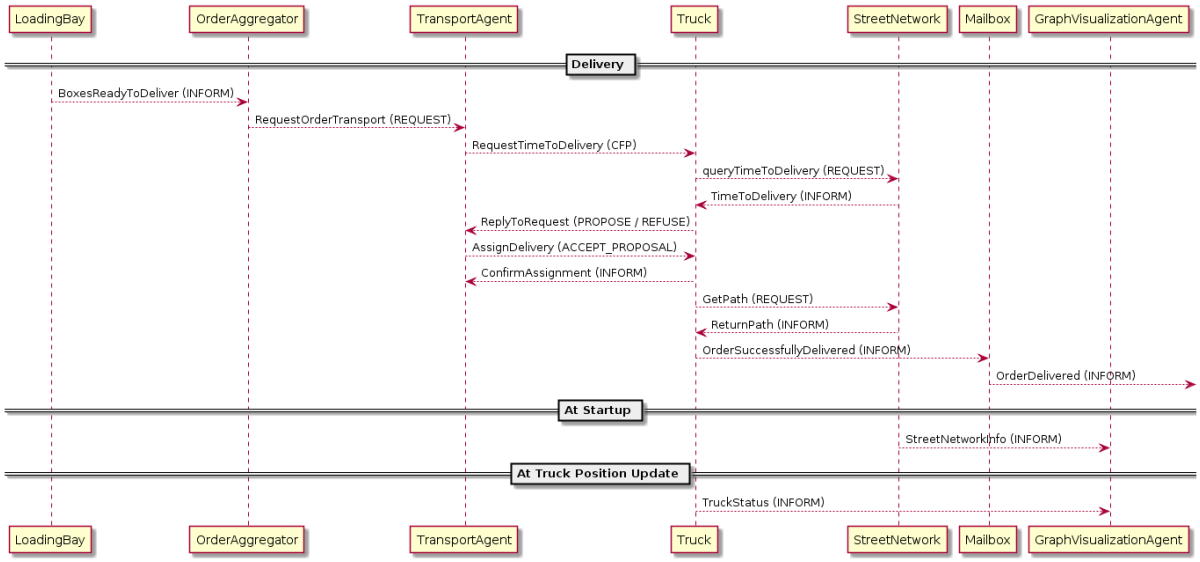

Figure 3: Delivery Stage Architecture

In order to find the best truck that can deliver the orders, the transport agent of the bakery requests for proposals from the trucks to deliver the order. Each truck which is capable of delivering this order then queries the StreetNetwork to find the time needed to complete its current task, return to the bakery to collect the products, and then move towards the customer. These estimates are then sent back to the transport agent, which then chooses the truck that has the least time to delivery. Section 3.1 provides a detailed description of all the agents that are a part of the delivery stage.

3.1 Agents

3.1.1 Order Aggregator (Developer: Abanoub)

Based on the idea that the loading bay will send the boxes of products that are complete, this agent is responsible for gathering all the needed information about the boxes sent by the loading

bay and forward the complete orders to the transport agent after it makes sure that the order is fulfilled. The agent works as follows: it keeps track of all the orders that the order processor broadcast. Then it waits until the loading bay sends boxes to be delivered. The boxes with their order id are used to check the order fulfillment and then send the full order to the transport agent.

3.1.2 Transport Agent (Developer: Abanoub)

This agent is responsible for receiving the boxes of fulfilled orders from the order aggregator, getting the location of the bakeries and the customers of this specific order, assigning the order to the trucks with the best delivery time and handing the orders to the trucks when they arrive at the bakeries. This is achieved as follows: It keeps updating a list of customers IDs associated with their orders IDs sent by the order processor. Then the cycle starts when it receives boxes from the order aggregator. It uses the details sent by the order aggregator to find the location of the customer and the bakery associated with this order. After that it communicates with all the trucks to call for proposals with the time each of them needs to deliver the order, then chooses the truck with the best time and assigns the order to it. When the truck arrives at the bakery to pick the order it gives the boxes to it.
Each transport agent is associated with a bakery.

3.1.3 Truck Agent (Developer: Sushant)

This agent is responsible for collecting orders from a bakery and delivering them to the customers. Each truck is associated with a transport company and multiple transport companies could exist. It is responsible for querying the estimated delivery time for an order and proposing this to the transport agent. If the proposal is accepted, it is also responsible for moving towards the bakery to collect the orders and then moving towards the customers to deliver them. Finally, it also informs the status of the delivery (such as the time of delivery, number of boxes) to the mailbox. Due to its highly dynamic nature, this agent has an extensive communication with other agents, as can be seen from the figure 3. It must be noted that with the current implementation, every truck can accept at most one extra order in addition to the one it is currently handling.

Additionally this agent also communicates with the GraphVisualizationAgent in order to send its position and status updates for visualization.

3.1.4 Street Network Agent (Developer: Ahmed)

The street network agent is responsible for maintaining the world 'map' in a graphical representation, containing the locations of all places, connections between them, and the distances, as specified in each scenario. Additionally, it provides navigational and temporal information to truck agents. This includes the paths that they must traverse to get from one location to another, and the time it would take to do so.

With the help of a few helper classes, such as *Graph*, *Vertex*, and *Edge*, the agent stores and updates (if necessary) the map as a directed graph. It makes use of an implementation of the Dijkstra algorithm that facilitates a graph search to find the shortest path between two nodes. This implementation is adapted from Lars Vogel's, which can be found in this link.

The agent communicates with truck agents and the graph visualization agent. The graph visualization agent constructs the visualization of the street network by obtaining the graph representation of the map from the street network agent.

3.1.5 Mailbox Agent (Developer: Ahmed)

The mail box agent is the interface agent between the final delivery stage and the customers. It receives the details of delivered orders from truck agents, and confirms these deliveries to the concerned customer agents. It can be extended to relay this information to any agent who requires it.

The agent sends a message which contains all the relevant details of a confirmed delivery, such as the truck (and delivery company) that delivered it, the customer, the time, and the contents of the delivery.

Message Descriptions:

(Incoming) Truck Message:

Description: The TruckAgent informs the mailbox about a delivered order using the following message:
Performative: INFORM
Sender: AID of the TruckAgent that sends this message
Receiver: AID of the MailboxAgent
PostTimeStamp: Current system time
Conversation ID: *orderID*

```
{
  "DeliveryStatus": {
    "OrderDeliveredTo": "Customer-001",
    "OrderDeliveredBy": "Truck-001",
    "DayOfDelivery": 35,
    "TimeOfDelivery": 16,
    "NumOfBoxes": 5,
    "ProducedBy": "Flying Saucers Bakery"
  }
}
```

Figure 4: Mailbox Example Message

Figure 4 shows an example of the contents of the message

(Outgoing) Fulfilled Order Boxes Message:

Description: The mail box relays the delivery status message to the concerned customer and other agents who are interested in knowing the delivery status of the order using the following message:
Performative: INFORM
Sender: AID of the MailboxAgent
Receiver: AID of the Concerned CutomerAgent. If more agents are interested in receiving this message, their AID's will also be added to the receiver list.
PostTimeStamp: Current system time
Conversation ID: *orderID*

The message is of the same format as the incoming message, and an example can be seen on Figure 4.

3.2 Objects

The *Graph*, *Vertex*, *Edge*, and *DijkstraAlgorithm* classes are taken from Lars Vogel's implementation of the Dijkstra algorithm. A link to the source code and an accompanying blog post can be found here.

3.2.1 Graph

The Graph object implements a directed graph of nodes and edges to be used in the Dijkstra algorithm. It maintains two lists: one consisting of *Vertex* objects, and another of *Edge* objects.

3.2.2 Vertex

The Vertex object represents a vertex in a directed graph to be used in the Dijkstra algorithm. The class maintains a vertex's, or node's, ID and name as strings, and a few helper functions. For example, an equality helper function that checks whether two vertex objects are equivalent.

3.2.3 Edge

The Edge object represents an edge in a directed graph to be used in the Dijkstra algorithm. Each edge stores a source vertex/node, a destination node, a unique ID, and a weight value.

3.2.4 DijkstraAlgorithm

The DijsktraAlgorithm class implements the Dijkstra algorithm for finding the shortest path between two nodes in a directed graph. This is used in the street network agent to enable returning of paths between two locations, as well as the time needed for the journey, to truck agents which as for this information.

3.2.5 Box

The box class characterizes the requirements of a box object that has been described in the project description. This class holds a fixed amount of products of a single product type. Box objects can be uniquely identified using the box id. This class also provides helper functions such as adding products to a box, querying its ID and printing the details of the box.

3.3 Graph Visualization Agent (Developer: All)

The graph visualization agent facilitates observing the delivery process of the orders in a graphical way. This agent is capable of displaying the entire street network (consisting of its nodes and edges) and all the trucks present at any given time as shown in figure 5. JavaFX was used to generate the nice display and we used and modified the implementation of Roland to generate the graph representation.

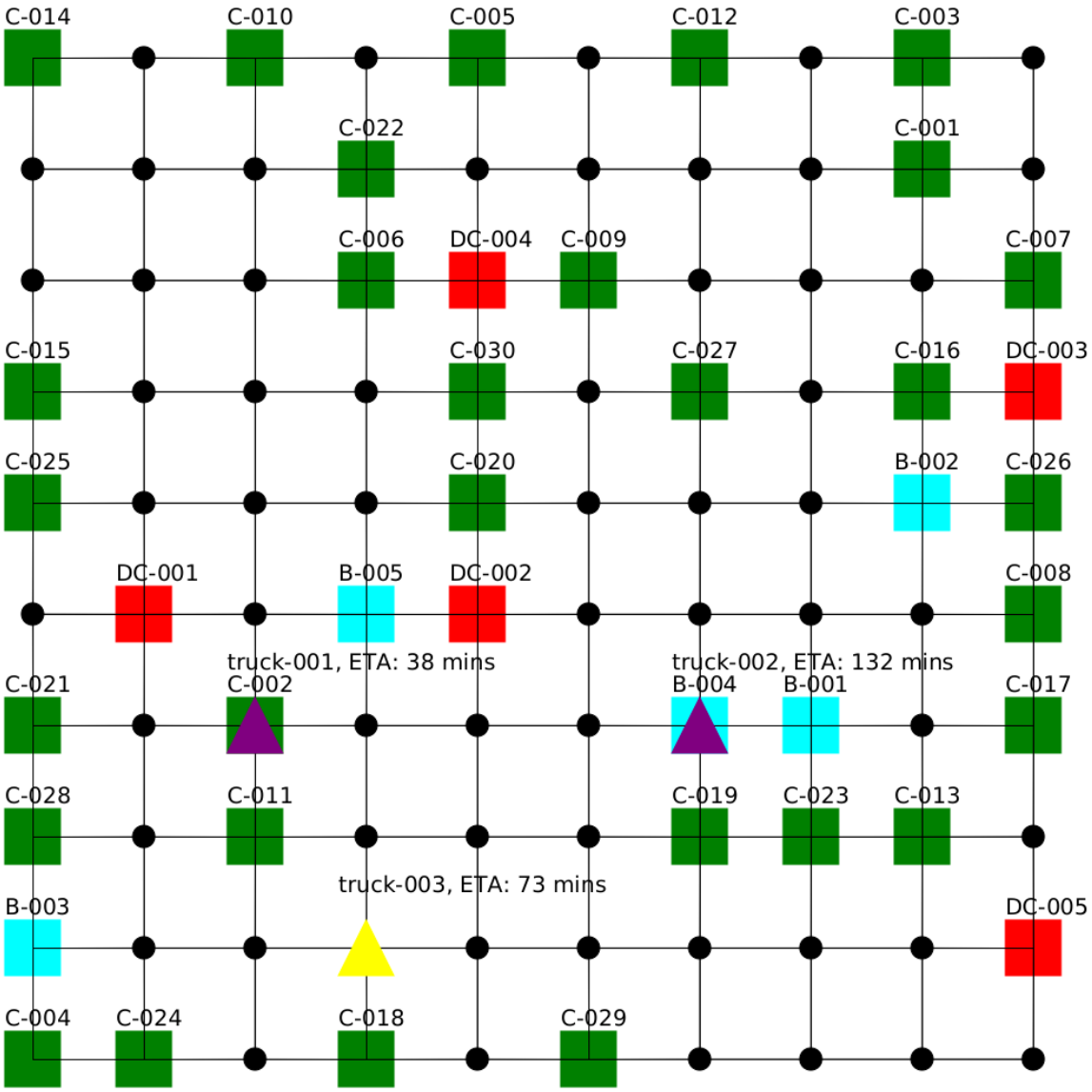

Figure 5: Screenshot of Delivery Stage Visualization

The nodes of the graph are displayed as either squares (for important nodes such as bakery or customer locations) or small circles. The nodes are then connected by the edges. The squares are further color coded in order to distinguish the type of node. Green represents customers, red delivery companies and cyan bakery locations. The trucks are represented using triangles on the graph and it is possible to update their location based on their actual position in the

graph at any time step. Similar to the nodes, the trucks are also color coded to identify their states. Gray represents idle state, yellow moving towards bakery for collecting boxes and purple delivering boxes to a customer. Additionally, the trucks also display the expected time of arrival along their current route.

This agent communicates with two agents in the delivery stage. The street network agent sends the street network information (consisting of the nodes and the edges) at startup to the graph visualization agent. Similarly, the trucks inform the graph visualization agent about their start positions. Whenever the position of a truck is updated, a new message is sent to the graph visualization agent informing it about the new position, expected time of arrival and state of the truck.

4 Usage Instructions

4.1 Running the two stages

In order to run the simulation of these two stages, follow the below steps:

1. Clone the repository from GitHub
2. Install the gradle version mentioned in the README of the cloned repository.
3. Ensure that the EntryAgent is active by checking the file *PackagingStageInitializer.java* by uncommenting the below line if it is commented.

   ```
   \textit{agents.add("EntryAgent:org.commitment_issues.packaging_agents.
   EntryAgent(" + scenarioDirectory + ")");
   ```

4. Start the program using the command

   ```
   gradle run --args='-packaging -delivery -visualization'
   ```

5. If you get an error saying gradle does not support '*args*'. Then use the below command instead to start the program

   ```
   ./gradlew run --args='-packaging -delivery -visualization'
   ```

4.2 Running on distributed systems

Follow these steps to run multiple stages on different systems.

1. Connect all systems to the same network.
2. Search for open ports using the command:

   ```
   netstat -lntu
   ```

3. Look for a port whose state is not LISTEN (this will be the PORT used below).
4. Find the host's IP address using the command: (in field wlp8s0, Inet addr: ...)

   ```
   ifconfig
   ```

5. Start the host using the command:

   ```
   gradle run --args='-isHost [host ip address] -localPort [PORT] -[STAGE] -noTK'
   ```

   Example:

   ```
   gradle run --args='-isHost 192.168.43.113 -localPort 5353 -packaging -noTK'
   ```

6. Start a client using the command:

   ```
   gradle run --args='-host [HOST IP ADDRESS] -port [HOST LISTENING PORT] -[STAGE]'
   ```

   Example:

   ```
   gradle run --args='-host 192.168.43.113 -port 5353 -packaging'
   ```