Multi-Agent Systems Coursework

Set 10111

Student: Darren Simpson

Matric: 40284701

Contents

1. Introduction

The goal of this project was to create a multi-agent system which deals with the smartphone supply chain problem. The system will take customer orders, purchase components in order to fulfil these orders, assemble the phones and then ship the finished products to the customers. The aim of this is to represent a Supply Chain and replicate the day to day activities of a company looking to maximise profits with daily changing variables.

The supply chain problem is increasingly important when considering the global economy. Supply chains are described as: “highly dynamic environments that are subject to: market fluctuations…Operational contingencies…changes in strategies” (*Arunchalam and Sadeh, 2005*). It is due to these constant changes that being able to evaluate and alter strategies is paramount when trying to create a thriving business. As a result, creating a system that can independently adapt to these changes can lead to success.

A Multi-Agent system is one which utilises autonomous agents in order to carry out tasks depending on both the behaviours with which it is programmed and the variables which it is given. This makes them well equipped when considering changing conditions in the marketplace. In the case of this project, the manufacturing agent will react to changing orders daily and decide which suppliers to use based on price and delivery time. This demonstrates the power of the multi-agent systems in a fluctuating market with constantly changing elements.

2. Model Design

The design consists of four types of agents: **Sync, Manufacturer**, **Customer** and **Supplier.** Each of these carry out their own daily activities and interact with one another in order to meet an end goal.

Sync: The Sync agent is responsible for sending ticks to each of the other agents in order to synchronise their actions on each day. It sends a “new day” message which *informs* each agent of a new day and prompts them to begin their daily activities. When it receives an *inform* message from all other agents saying they are “done” with their activities, it will send a final *inform* that tells all agents to “terminate” and that the day is done. The inform is a useful protocol for this as no reply or request is required, simply sending a message to trigger. *(appendix 2)*

Manufacturer: The manufacturer agent has the most responsibilities of all the agents. It is the job of the manufacturer to decide on which incoming orders to accept from customers, followed by sending a request to the suppliers for parts to assemble the order, preferably by the due date. Once it has done this, it will then check to see if it has the available resources to build any orders. If not, then it will wait for the next day in which it will collected delivered components and repeat its tasks for the day once more. Each day it will generate its total cost to show the profits.

The manufacturer has its own ontology used to identify it and two other related ontologies, *ManufacturerDeliveryOwns* and *ManufacturerOwns*. As you can see in *appendix 1,* these are used to connect between order ontologies and a list of component ontologies in order to receive orders or to receive incoming deliveries from suppliers.

The manufacturer communicates with the *SyncAgent* in the same way as all others, by sending inform messages say it is “done”. It also uses a **request** in order to pass a message to the supplier, requesting that they populate a list with the required components to be sent after the set delivery time. This request allows for the supplier to use *MatchPerformative* to return an ACL Message which matches the type of the message sent.

Supplier: There are two suppliers in the system, each which have a different set of components with different costs and delivery times. The role of the supplier is to wait until contacted by the customer and then generate a delivery based on what is requested from them. Depending on which supplier, they will either send the delivery the next morning or send it in the stated amount of days.

The supplier ontology is similar to the manufacturer. It has its own communication ontology which includes the agent name and sets an instance of supplier. It has two other related ontologies, *SupplierOwns* which is Predicate and contains the supplier Agent ID and the order of the manufacturer and *SupplierResponseOwns,* also Predicate, which contains the manufacturer Agent ID and a list of Component Ontologies used for sending the requested parts to the manufacturer. These both allow for easy communication and the passing of all the information needed to receive the order and return what is required. (*appendix 1)*

The supplier communicates only with the *SyncAgent* and the Manufacturer agent. As stated with other agents, it sends an inform message to the *SyncAgent* in order to state when it is “done” with its tasks and receives inform messages to tell it of a new day or the end of a day. When communicating with the manufacturer, as stated in the manufacturer section, it uses *MatchPerformative* when receiving the order in order to match the type of the message sent. It also sends its own request to manufacturer when returning the list of components that have been requested.

Customer: The role of the customer agent is to generate an order based on a formula which creates a random device from a list of possible components. This device will then be given a random price, due date, quantity of devices and late fees and sent to the manufacturer in order to be processed. The number of customer agents can be increased in the application, however the default that will be running is 3. Each of these agents shares the same task of generating a device with details and sending it to the manufacturer.

Customer ontology is the most basic of the four and uses the same communication ontology stated above in order to set an instance of customer. The customer ontology also uses an Order Ontology when sending a generated device in order to the manufacturer. This is useful as it keeps the orders uniform, allowing for all agents to know what should be included within an order.

For communication, it uses the same **inform** when communicating with the *SyncAgent* as all other agents. It's only other form of communication is a **request** which is used when sending the generated order which included the *ManufacturerOwns* ontology when specifying the receiver and the content of the message. (*appendix 1)*

3. Model Implementation

Each of the agents operates on their own set of behaviours each day. In the case of manufacturer, it uses a sequential behaviour (named *dailyActivity*) which contains sub-behaviours which are carried out. These are made up predominantly of one-shot behaviours, meaning they are activated once per day and one cyclical behaviour which loops based on 'ticks' of time. ***TickerWaiter*** is cyclical and is used for receiving inform messages from the *SyncAgen*t. It runs throughout the activity cycle of the application and adds and removes behaviours at the start and end of the day respectively.

The second behaviour to run is ***CollectDelivery*** (one shot), this receives the list of components that has been sent the day before by the supplier. It is also responsible for generating the cost of components which is then later removed from the gross profit (*appendix 3.6, calculating cost for components*). Once the delivery is received, the warehouse stocks are incremented with what has been delivered.

Once the delivery has been collected, the manufacturer collects the orders from customers, this is also a one-shot behaviour. After the orders are collected, the manufacturer will attempt to assemble any orders which it has the available components for in ***AssembleAvailableOrders*** (one shot). It does this by first checking if the warehouse is empty. If it is, then this step is ignored entirely and one day is added to the order wait times. If the warehouse is not empty, it will check to make sure there are enough required components to complete the order. If there are, it completes the order and the profit is updated, if not a message is displayed, and a day is added to the order wait times (*appendix 3.2*). It is within this behaviour that the late fees are generated by checking days waited for the completed order against the number of days due. If the days overdue is greater than the due date,

then the agreed late fee is charged per day late (*appendix 3.3*). This is then applied to the final profit (*appendix 3.6).*

The final behaviours of the day is to ***find the suppliers*** (one shot), ***send the component order*** for the next days deliver (one-shot) and finally the ***end day*** behaviour (one-shot) which is where the warehouse costs per item is generated (*appendix 3.4*) and the final profit for the day is generated (appendix 3.6). It then sends it's "done" message to the *SyncAgent* to complete the day.

The other agents have fewer behaviours per day. Customer has the same cyclical ***TickerWaiter*** as customer, however the only behaviours which are added is ***generateOrder***(one shot) and ***EndDay*** (one-shot). Once the day has been started, the customer uses an assembler class which uses *math.rand* in order to generate random numbers in order to decide on the components of device. First it decides on the size of the phone, either setting the screen and battery to small (smartphone) or to large (phablet). It will then carry out the same process for the size of memory and storage, however these are not tied to the size of the phone. (*appendix 3.1)*. Once the device is generated, it is sent to the manufacturer and end day is called and a "done" message is sent to the *SyncAgent*.

The supplier agent once again operates on a ***TickerWaiter*** to prompt the beginning and end of the day. It starts by finding the manufacturer (one shot) in order to receive an order. Once an order is sent, the supplier gets the specification for the device and the quantity of devices needed and uses this to generate a list of components. From here it will send this to the customer either on the next day for *supplier1* or in four days time for *supplier2.* This is set within the ontology of the components and is used in the *MainContainer* when the components are set (*appendix 3.5)*. Once the order has been completed, the supplier sends a "done" message through the end day behaviour to the *SyncAgent* to complete say it has completed it's day.

4. Design of Manufacturer Agent Control Strategy

Unfortunately, I was unable to implement the Manufacturer Agent Control Strategy in my system. However, there was a planned strategy that will be outlined within this section.

Deciding Customer Orders:

When a customer order is accepted, it would be added to a list of *customerOrders* which would contain the AID of the customer and the order. From there, there would be a loop to find the amount of orders in the list. First of all, it would count the number of components required for each order and would add any orders which could immediately be created to a list of *chosenOrders*. A further loop would then take the first quantity of the order and compare it to the index of the next order's quantity in the list. If these quantities were less than 50 (the maximum amount of orders per day), it would be chosen as the best combination of orders. It would then continue the loop and if any of the orders

remaining would keep the max orders below 50, they would be added. The profit for this set of orders would then be generated and set as the *bestProfit*. There would then be a second loop which would generate a list of *possibleOrders* with the remaining combinations of orders. If any of these combinations produced a greater profit then they would replace the previous selection of *bestProfit*.

How Many Components to Order:

Once a decision has been made on which orders to accept, the system would have to decide both how many components to order and from which supplier. I would carry this out by comparing the required components to what is currently available in warehouse. I would first find the number of screens and batteries that are required and order them from supplier1 as it is the only supplier with those components available. I would then check to see any of the screens or batteries were below a set number (I feel 40 would be suitable, though ideally would have liked to have tested in section 5 with this variable) and if so, I would order the number of components between the current amount and 40, with a minimum of 10.

I would then find the *dueDate* on the orders and on any that had 5 days or more, I would order the components from Supplier2 as it would be cheaper to order the batteries and memory without incurring the late fee. It would also lead to less parts stored in the warehouse which would hopefully lower costs. On orders which would take over the 4 days for delivery, I would add together the *lateFee* for the order and the price for components from supplier2 and compare it to the price of components from supplier1 and the cost for storing that many parts (quantity of components * 5) and whichever cost loss, that would be the option I chose.

5. Experimental Results

As stated before, I was unable to complete the control strategy, however with what I do have I am able to check two main variables and see how they affect the overall running of the system. These variables are: **Amount of Customers** and **Price for Warehouse Storage.**

I feel that I can only make so many hypotheses on what will happen due to the limited scope of testing; however, I will still be interested to see how these variables change the overall profit. I suspect that by increasing the amount of customers, the system will initially lose profits due to the cost of ordering (*x)* more components and as such, also storing more in the warehouse. I suspect that when increasing this the price will increase exponentially and outweigh the cost of components and storage.

As for the cost of warehouse storage, this feels like it will be a very clear hypothesis to make. Due to the simplistic nature of the agent, it will accept all orders and create them on the next day, therefore there isn't a large amount of storage used at any given time. I expect to see a fairly unnoticeable change in costs when decreasing the cost, however with the increase I will be curious to see how much it does increase due to storing large amounts of components in bulk daily.

Increasing Customer Number:

Each of the tests were run five times and a mean was found of all 100 end day totals

Surprisingly, when increasing the customer number, the mean figure turned out to be fairly close when considering changing from 1 to 3 customers. I had assumed that with extra customers it would increase component and storage costs, but I had also expected a much higher mean gain overall.

Even more surprising was that increasing the number of customers to 5 actually dropped the mean total by a large margin. I believe this to be because of the high hit the company takes in ordering components and the amount of money being spent daily to store them. This is what I had expected, but I had also assumed that due to more orders being completed, more profit would be generated, and I would see an upward climb in sales. Perhaps that would have been the result if I had run the simulation for more than 100 days.

Figure 1 Additional Customer Graph

Increasing and Decreasing Warehouse Cost:

Each of the tests were run five times with 3 customers and a mean was found of all 100 end day totals

Due to 5.0 being the default used in the program, I used that as the basis for what I expected based on the customer results. The results ended up being a little confusing for me. I had expect a clear curve of profits when the warehouse price was low and high. What happened however was that the difference between 3.0 and 10.0 was not as severe as expected and the default, 5.0 actually turned out to be the highest grossing. I suspect this is due simply to the random nature of the orders and the lack of decision making based on the part of my manufacturing agent.

Figure 2 Warehouse Cost Increase

6. Conclusion

While overall pleased with the work I have completed, it has to be said that my particular multiagent system is lacking when considering a control strategy, so when looking to how it could be improved, that would obviously be the first place I would think. With that implemented however, I believe that it could be a useful tool in completing it's given task. This does not translate to the real environment that it is aimed at however. With constantly changing variables and ever evolving requirements, it can be difficult to create a system that can react and maximise profits effectively.

The first change I would consider making would be a way to monitor prices of components. This would be useful so that when a particular component drops to a lower price, the system could decide to stock up and likewise when the price increases it could lower the amount of orders.

Another way to improve to match a more realistic scenario would be to generate analytical data to track components being included and components going unused in the warehouse. This would lead to being able to alter purchasing supplies based on what is selling out quicker and not making broad orders for components that are used less frequently. This would also help when considering the cost of warehouse storage, though I am sure in a more realistic scenario this would not change based on individual items.

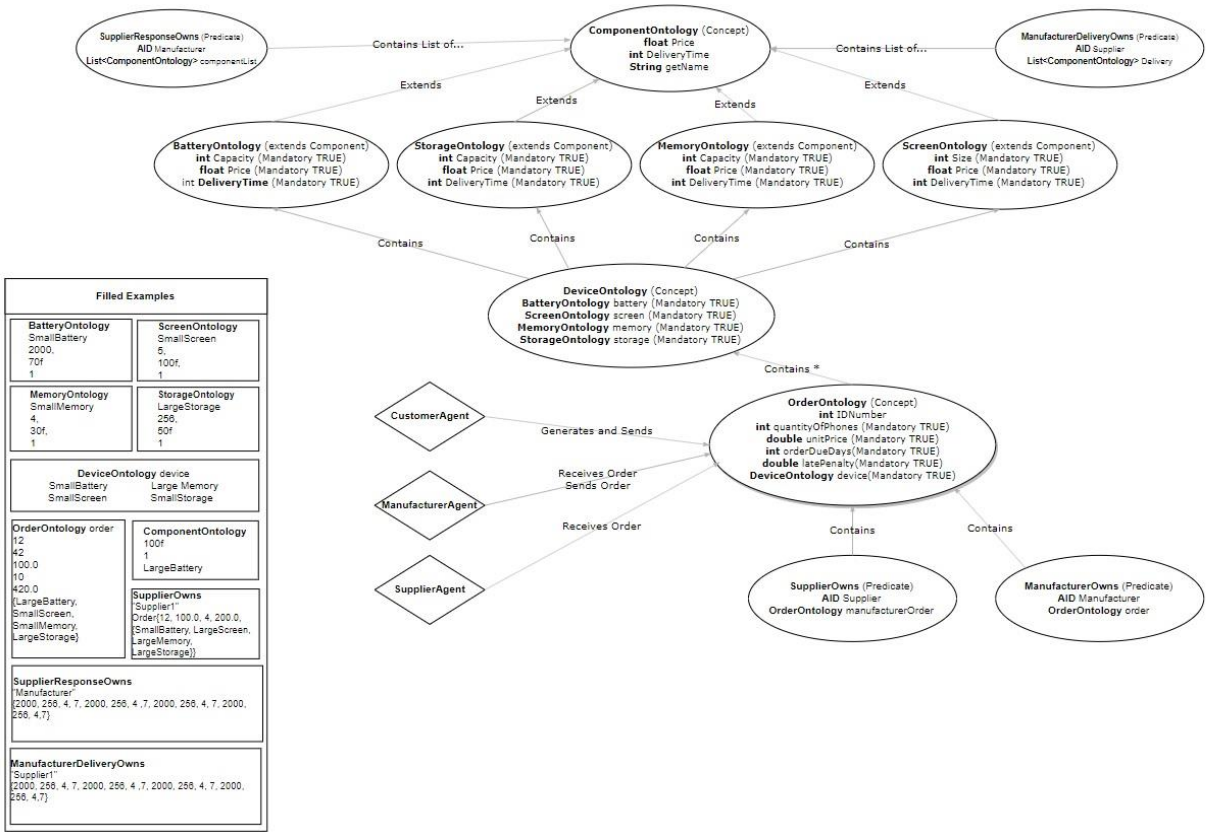
Overall, I am happy with what my current system includes but feel with just a little more time I would have been able to complete 100% of the brief and I am incredibly disappointed that I would not get a functional agent control strategy. If I were to improve the proposed system, I would prioritise orders based on days until order due and the cost per late day. This would mean that orders which would run over time but cost less in late fees would be put aside in order to focus on orders which would have higher late fees.

I feel that the advantage of using this kind of system is that there are so many areas that you can modify and analyse in order to find where profits are being lost and gained. The ability to simulate as many days as needed and have the agents autonomously create data is extremely powerful and it's clear how it could be used effectively in a real life system.

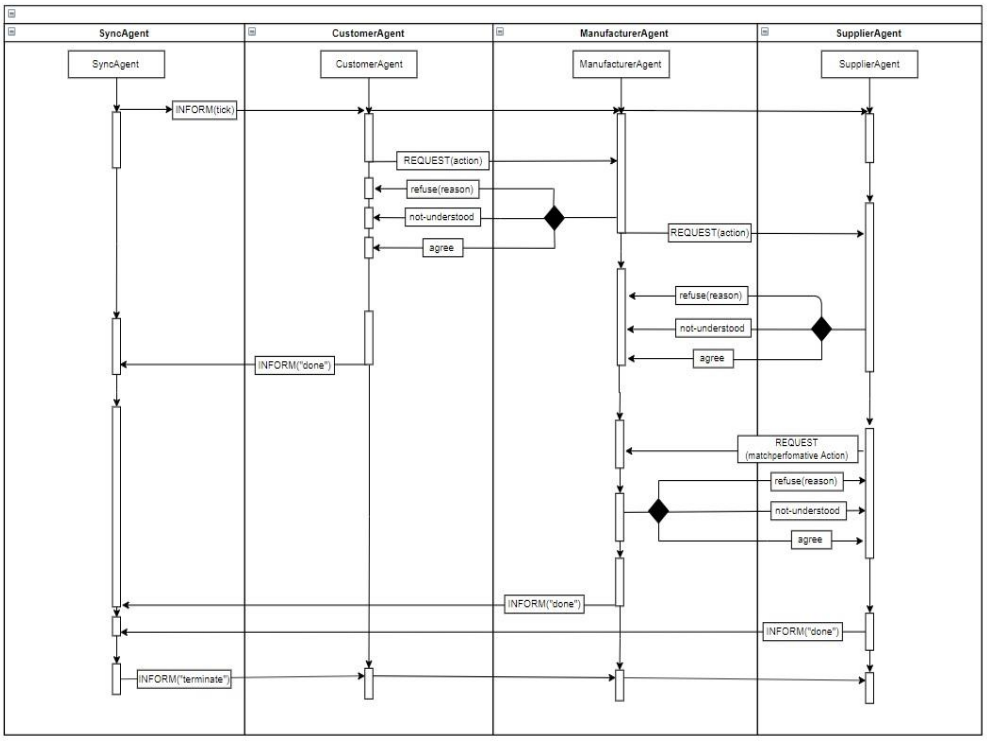
References

Arunchalam, R. and Sadeh, N. (2005). The supply chain trading competition. *Electronic Commerce Research and Applications*, 4(1).

Appendix 1: ontology



Appendix 2: Communication Protocols



FIPA SL EXAMPLES:

(**Inform-tick**
:**Sender:** (agent-identifier : Sync Agent)
:**Receiver:** (set (agent-identifier : CustomerAgent
ManufacturerAgent
SupplierAgent)
:**Content** ("new day")
:**language** fipa-sl
)

(**Request**
:**sender** (agent-identifier : CustomerAgent)
: **receiver** (set (agent-identifier : ManufacturerAgent)
:**Content** (Screen: 7" Battery: 3000mAh Memory: 4GB Storage: 256GB | Quantity:20 Price:282.0 Days Until Order Due:5 Cost per day late:900.0)
:**language** fipa-sl
)

(**Request**
:**sender** (agent-identifier : ManufacturerAgent)
: **receiver** (set (agent-identifier : SupplierAgent)
:**Content** (Screen: 7" Battery: 3000mAh Memory: 4GB Storage: 256GB)
:**language** fipa-sl
)

(**Request**
:**sender** (agent-identifier : SupplierAgent)
: **receiver** (set (agent-identifier :ManufacturerAgent)
:**Content** (7", 3000, 4, 256, 7", 3000, 4, 256, 7", 3000, 4, 256, 7", 3000, 4, 256, 7", 3000, 4, 256, 7", 3000, 4, 256) - ***example of sending back parts for 6 phones***

:**language** fipa-sl
)

(**Inform-done**
:**Sender:** (agent-identifier : CustomerAgent,
ManufacturerAgent
SupplierAgent)
:**Receiver:** (set (agent-identifier : Sync Agent)
:**Content** ("done")
:**language** fipa-sl
)

(**Inform-terminate**
:**Sender:** (agent-identifier : Sync Agent)
:**Receiver:** (set (agent-identifier : CustomerAgent
ManufacturerAgent
SupplierAgent)
:**Content** ("**terminate**")
:**language** fipa-sl
)

Appendix 3: source code

```
if(Math.random() < 0.5) {
      // Small smartphone
      screen = new ScreenOntology(5);
      battery = new BatteryOntology(2000);
   } else {
      // phablet
      screen = new ScreenOntology(7);
      battery = new BatteryOntology(3000);
   }
```

Figure 3 ensuring small phone or phablet

```
//check to see if warehouse is empty, if yes, add 1 day to days waited
System.out.println("Warehouse is empty");
            for (OrderOntology collectedOrder : collectedOrders) {
               collectedOrder.setDaysWaited(collectedOrder.getDaysWaited()+1);
            }
            return;
         }

//if not enough components to assemble phones, add to days waited
else {System.out.println("Not enough components to assemble phone, awaiting next delivery");
    collectedOrder.setDaysWaited(collectedOrder.getDaysWaited()+1);}
```

Figure 4 sufficient components in warehouse

```
double totalLateFee = 0;
               int daysOverdue = collectedOrder.getDaysWaited() - collectedOrder.getOrderDueDays();
               if(daysOverdue > 0) {
                  totalLateFee = collectedOrder.getLatePenalty() * daysOverdue;
               }
```

Figure 5 generating late fees

```
 int warehouseStock = Warehouse.size();
        double warehouseCost = warehouseStock * perItemCost;
        System.out.println("Daily Warehouse Stock Cost: £" + warehouseCost);
```

Figure 6 Per-component per-day warehouse cost

```
ComponentOntology components1[] = {
            new BatteryOntology(2000, 70f, 1) //final value is delivery time


ComponentOntology[] components2 = {
            new StorageOntology(64, 15f, 4) //final value is delivery time

//component ontology declaring deliverytime
public class ComponentOntology implements Concept {
   private float price;
   private int deliveryTime;
   private String name;


//getters and setters for deliverytime
public int getDeliveryTime() {
      return deliveryTime;
   }

   public void setDeliveryTime(int deliveryTime) {
      this.deliveryTime = deliveryTime;
   }
```

Figure 7 delivery time for suppliers

```
//calculating cost for delivery
totalProfit -= totalCost;
System.out.println("Total cost: £" + totalCost + " for " + collectedDelivery.size() + " Components");

//calculating profit minus the late fee
totalProfit += totalGain - totalLateFee;

            System.out.println("Sold " + quantityNeeded + " phones for £" + pricePerPhone
                  + " Total Gain: £" + totalGain + ". Late Fees: £" + totalLateFee);


//calculating total profit minus warehouse costs
System.out.println("Daily Warehouse Stock Cost: £" + warehouseCost);
         System.out.println("Manufacturer Profit Updated!: £" + (totalProfit - warehouseCost));
```

Figure 8 calculating final profit