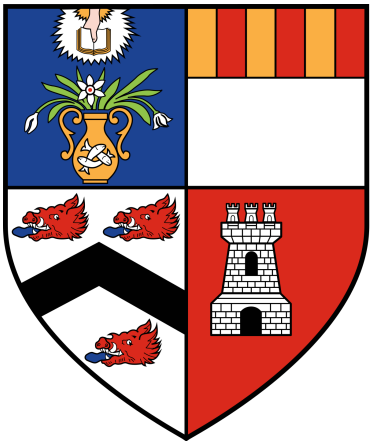
CS551K - Software Agents & Multi-Agent Systems

In-course Summative Assessment

Ricardo Soares

MSc Artificial Intelligence



Department of Computer Science
University of Aberdeen
4^{th} of April 2018

Contents

1 Design

1.1 AUML

1.1.1 AUML Diagram

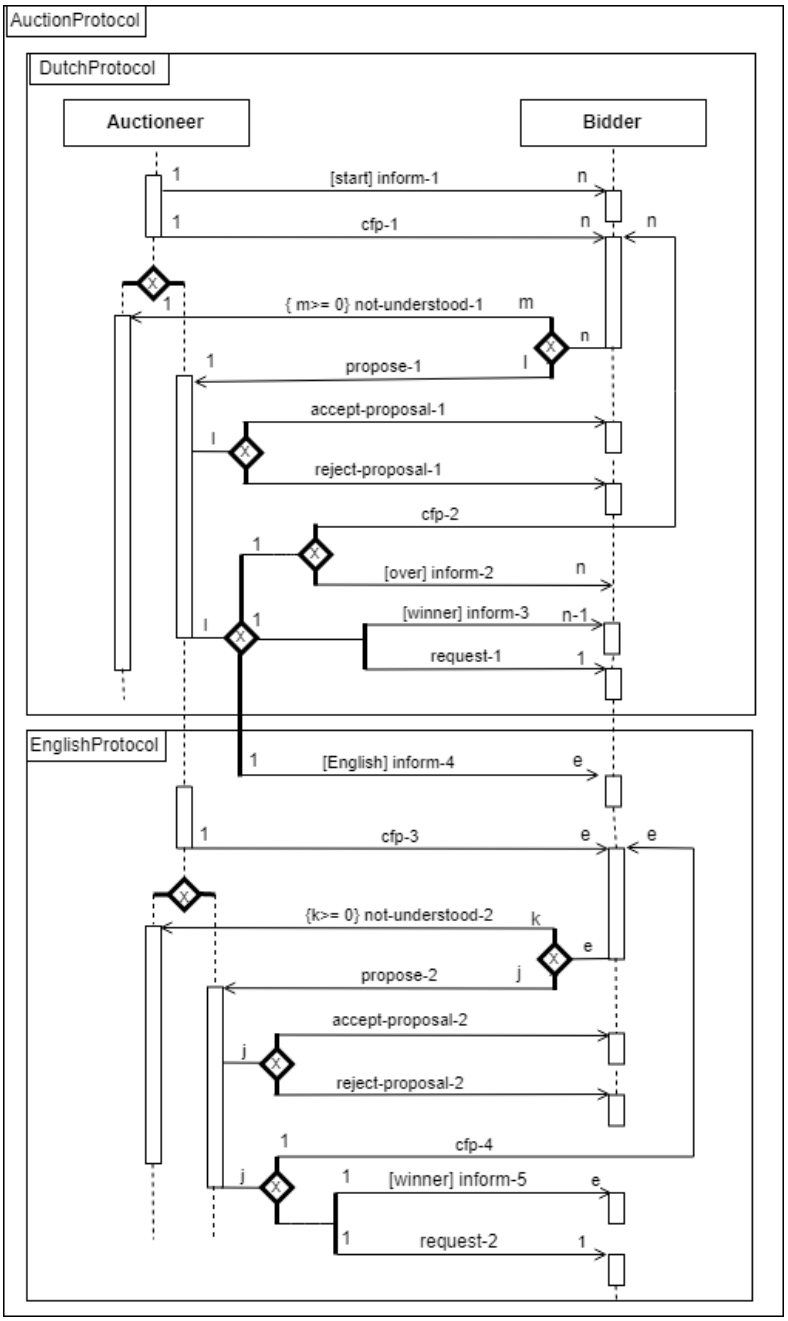


Figure 1: AUML diagram.

The *AUML diagram* represented above illustrates the workflow between the **Auctioneer** and a set of **Bidders**. The *DutchProtocol* and *EnglishProtocol* boxes are depicted with illustration in mind, to permit the reader to better understand the two main, different sections of the *AuctionProtocol*. First, to explain the collection of messages sent through the diagram, the possible message types will be identified, followed by the understanding of the individual messages and connections.

1.1.2 Message Types

To start, it's relevant to consider the arguments each *Message* has. Considering the *Auction environment* this assessment is presented in, certain message types will require added arguments; yet, every message will share *core arguments* as part of its basic structure. These will be omitted as the different message types are compared.

Without further ado, these core arguments are:

- **Message Type** - To understand how to process the message as it is received, the message type is vital.
- **Sender Identifier** - The Sender must be explicit, so the Receiver of the message knows who sent it.
- **Receiver Identifier** - Without properly identifying the Receiver, the message will, obviously, not reach its destination (*e.g. a postcard without destination*).
- **Auction Identifier** - Albeit domain specific, every message must refer to the Auction it relates to. It is, briefly put, the topic that gives purpose to the communication.

After identifying the core arguments, the extra parameters certain message types declare will be presented with their description. Keep in mind that certain message's parameters are only exposed in the AUML diagram when it's deemed as necessary; as such, these parameters will aid the implementation more than the AUML . There are **seven** possible *message types: "not-understood", "call-for-proposal", "inform", "propose", "accept-proposal", "reject-proposal"* and *"request"*. Their use is represented as follows:

- *"not-understood"* - The Agent did not comprehend the previous communication. There are no extra parameters needed.
- *"call-for-proposal"* - The Agent sends a call for proposals to other agents. In this *Auction environment*, "call-for-proposal" is used by the **Auctioneer** to inquire the **Bidders** for **bids**. Hence, there are two extra parameters: The *Good*, and the *Price*.
- *"inform"* - The Agent informs other agents of a certain piece of information. The extra parameter needed would be the content of such information.
- *"propose"* - The Agent replies to the *"call-for-proposal"* message with his proposal. The extra parameter in our *agent environment* the value of the Offer; the Good is omitted.

- *"accept-proposal"* - The Agent accepts the proposal of another agent. In our *Auction environment*, it's deemed as relevant to understand the *"accept-proposal"* means the bid was accepted, yet this translates as the **best of the round**. It does **not** translate to **winning the Auction**. Its extra parameter is the proposal's Offer.

- *"reject-proposal"* - The Agent rejects the proposal of another agent. In our domain, it represents that the bid was not sufficient. There are no extra parameters.

- *"request"* - The Agent requests for an action to be performed by another Agent. According to the *Auction environment*, this translates as the demand for payment, for the afforded good. It also implies to the receiving Agent that his previously accepted *"propose"* message had won the Auction winning round.

From these *message types*, the distinction between the *"accept-proposal"* and *"request"* is the less intuitive to grasp, due to the interpretation provided to these messages. As such, the comprehension of these is vital to the proceedings of this report.

1.1.3 Message Flow

To clear any doubts of the proposed AUML diagram, the messages will have its instances briefly explained, according to the work flow presented. These are as follows:

- *"[start] inform-1"* - The **Auctioneer** informs every **Bidder** (n) of the auction's "start", which will begin as a *Dutch Auction*.

- *"cfp-1"* - The **Auctioneer** calls for the **Bidder**'s proposals. The message transmits the *Good* and the *Price* to match.

- **First mutually exclusive message flows:**

  1. *"{$m \geq 0$} not-understood-1"* - A set of m **Bidders** did not comprehend the "call-for-proposal". This could be a consequence of a missed "inform"; if the *Bidder* doesn't receive the "inform-1" message, he will not know about the auction referred in the "cfp-1".
  2. *"propose-1"* - The **Bidder** proposes a bid for the *Dutch Auction*.

- **Second mutually exclusive message flows:**

  1. *"accept-proposal-1"* - If it occurs, the **Bidder**'s proposal was the winning bid of not only the *Dutch round*, but also of the *Dutch Protocol*, since the latter only contains one bidding round. If the *English Auction* does not ensue, this **Bidder** will be the Auction's winner.
  2. *"reject-proposal-1"* - The **Bidder** has its proposal rejected. According to the scenario at hand, if this message occurs it is implied there was more than a bid in the *Dutch Auction*. Since every proposal has the same *Offer*, one random **Bidder**'s proposal is accepted, and the rest are refused.

- **Third mutually exclusive message flows (three branches):**
  1. **First branch, mutually exclusive. Covers situations regarding no proposals. (l = 0)**
     (a) *"cfp-2"* - The **Auctioneer** continued to a new round in the *Dutch Auction* with a new "call-for-proposal", after successfully reducing the bidding price.
     (b) *"[over] inform-2"* The **Auctioneer** refuses to reduce the bidding price, and the auction is terminated, unsuccessfully.
  2. **Second branch, both message flows occur. Covers situations regarding one proposal, which concludes the *Dutch Auction* with no *English Auction*. (l = 1)**
     (a) *"[Winner] inform-3"* - The **Auctioneer** informs every **Bidder** besides the winner (n-1) of the auction's positive result.
     (b) *"request-1"* The **Auctioneer** requests the money offered by the *Bidder*, finalizing the Auction without proceeding to the *English Protocol*.
  3. *"[English] inform-4"* - The **Auctioneer** informs the *Dutch Auction*'s last round **Bidders** (e) of the start of the *English Auction*, as only these will be allowed to join. This message depicts the beginning of the *English Auction*, and it covers the branch where there is more than one **Bidder** proposal ($l > 1$).

- *"cfp-3"* - The **Auctioneer** proceeds to the first "call-for-proposal" in the *English Auction*.

- **First mutually exclusive message flows in English Protocol:**
  1. *"{k≥0} not-understood-2"* - A set of k **Bidders** did not comprehend the "call-for-proposal", just as in the previous message with the same *message type*. Similarly as before, this would imply the **Bidder** was not informed of the *English Auction*'s start.
  2. *"propose-2"* - The **Bidder** proposes a bid for the *English Auction*.

- **Second mutually exclusive message flows in English Protocol:**
  1. *"accept-proposal-2"* - Contrary to the *Dutch Auction*, every round in the *English Auction* has an accepted proposal. The **Bidder**'s proposal was the winning bid of the *English round*.
  2. *"reject-proposal-2"* - The **Bidder** has its proposal rejected. This occurs to every **Bidder** except the one with the *Highest bid*.

- **Third mutually exclusive message flows in English Protocol (two branches):**
  1. *"cfp-4"* - The **Auctioneer** continued to a new round in the *English Auction* through a "call-for-proposal" with a higher price. Covers the branch where there is more than one **Bidder** proposal ($j > 1$).
  2. **Second branch, both message flows occur. covers situation where there were no proposals, which concludes the *English Auction* with the bid of the last *Round Winner* (j = 0):**

(a) *"[Winner] inform-5"* - The **Auctioneer** informs every **Bidder** besides the winner (n-1) of the auction's positive result.

(b) *"request-2"* The **Auctioneer** requests the money offered by the *Bidder*, finalizing the Auction and the *English Protocol*.

The abstract idealization of our **Auctioneer** and **Bidder**'s behaviours generate a sturdy groundwork for the steps to follow. We now proceed to the pseudo-code, where the realization of the initially designed protocols will be contrived to a higher degree.

Certain connotations were slightly tweaked to further imply the idea of the present domain. These will be referenced when exposed. The most prominent example would be how the *Messages* are also referred to as *Shouts*, for the rest of the assessment. Of course, every modified designation does not deviate from the semantic purpose of the original construct, careful enough to not confuse the reader, or put at risk the legibility of the work.

1.2 Pseudo-code

1.2.1 Auctioneer's Pseudo-code

Algorithm 1 Pseudo-code representing the Auctioneer's behaviour.

```
1: AuctioneerProtocol( ):
2:
3: Belongings ← {(Good, InitialPrice, Reserve, Increase), ...}   /*Goods to sell.*/
4: MyId                                                          /*Auctioneer's Id.*/
5: Credit                                                        /*Auctioneer's money.*/
6: for each (Good, Price, Reserve, Increase) ∈ Belongings do
7:     AuctionId ← AuctionId++          /*Next Auction will have the following Id.*/
8:     Participants ← {PId, ...} − MyId      /*Participants to attend the auction.*/
9:     (Bidder, bid) ← ∅              /*Obs: Winner of Round will be stored here.*/
10:    for all PId ∈ Participants do
11:        SendShout(inform, MyId, PId, AuctionId, "Start")      /*Start Auction*/
12:        SendShout(call-for-proposal, MyId, PId, AuctionId, Good, Price)
13:    end for
14:    AuctionOn ← True
15:    while AuctionOn do
16:        Shouts ← Get "Proposals" with correct AuctionId and Offer = Price
17:        if Shouts = ∅ then
18:            price ← price − (price ∗ increase)
19:            if price < reserve then        /*Obs: Only need to check reserve here.*/
20:                AuctionOn ← False                          /*Price is too low.*/
21:                for all PId ∈ Participants do
22:                    SendShout(inform, MyId, PId, AuctionId, "over")
23:                end for
24:            else                                  /*Next round. Call for Proposals!*/
25:                for all PId ∈ Participants do
26:                    SendShout(call-for-proposal, MyId, PId, AuctionId, Good, Price)
27:                end for
28:            end if
29:        else                                  /*English Auction or Single Dutch Bidder.*/
30:            DutchWin ← Shouts.Get(0)              /*Quickest / only proper bid.*/
31:            (Bidder, Bid) ← (DWId, DWOffer)     /*Wins if Shouts=1 or no Eng.*/
32:            AuctionOn ← False
33:            if Shouts > 1 then                    /*English Auction will ensue.*/
34:                Increase = Increase/4            /*Change price at a slower rate.*/
35:                Englishmen ← Get "Bidders" from Shouts
36:                for all EId ∈ Englishmen do
37:                    SendShout(inform, MyId, EId, AuctionId, "English")
38:                    SendShout(call-for-proposal, MyId, EId, AuctionId, Good, Price)
39:                end for
40:                EnglishOn ← True
41:                while EnglishOn do
42:                    Shouts ← Get "Proposals" with correct AuctionId, Offer > Price
```

```
43:                 if Shouts ← ∅ then
44:                     EnglishOn ← False                    /*No one wants to bid.*/
45:                 else
46:                     WinningBid ← Get "Proposal" with Highest bid from Shouts
47:                     (Bidder, Bid) ← (WBId, WBOffer)
48:                     Shouts ← Shouts−"WinningBid"          /*Shouts to refuse.*/
49:                     SendShout(accept, MyId, Bidder, AuctionId, Bid)
50:                     for all (proposal, EId, MyId, AuctionId, Offer) ∈ Shouts do
51:                         SendShout(refuse, MyId, EId, AuctionId, Offer)
52:                     end for
53:                     if Bid > Price + (Price ∗ Increase) then /*Next round price.*/
54:                         Price ← Bid
55:                     else
56:                         Price ← Price + (Price ∗ Increase)
57:                     end if
58:                     for all EId ∈ Englishmen do
59:                         SendShout(call-for-proposal, MyId, EId, AuctionId, Good, Price)
60:                     end for
61:                 end if
62:             end while
63:         else
64:             SendShout(accept, MyId, Bidder, AuctionId, Bid)     /*Dutch Win.*/
65:         end if
66:                 /* Point of Situation: (Bidder,Bid) populated with a winner.*/
67:                                     /*Impossible to reach without a winner.*/
68:                         /*After DWOffer and the "if" that leads to English Loop.*/
69:
70:         Participants ← Participants − Bidder
71:         SendShout(request, MyId, Bidder, AuctionId, Bid)      /*Request pay.*/
72:         Credit ← Credit + Bid
73:         SendGood(Good, Bidder)       /*Send Good to Bidder, according to Id.*/
74:         for all PId ∈ Participants do
75:             SendShout(inform, MyId, PId, AuctionId, "over")
76:             SendShout(inform, MyId, PId, AuctionId, "Bidder")
77:         end for
78:     end if
79: end while
80: end for
81: Switches to BidderProtocol( )
```

1.2.2 Auctioneer's Discussion

Regarding the pseudo-code concocted for the implementation of the *AuctioneerProtocol()*, certain assumptions and decisions were taken in consideration.

- The *AuctionId* is obtained from a static variable every *Auctioneer* will have the access of, which will increment as each auction initializes. The *AuctionId* is, therefore, unique.

- The *Belongings* are composed by a structure, with the referred variables *(Good, InitialPrice, Reserve and Increase).* Although the structure's denomination isn't defined here, it would represent the information relevant to the **Auctioneer**. Therefore, the **Bidder** will have a different perspective regarding the items he wants.
- *Participants* will be initialized as the full set of bidders, excluding, obviously, the Auctioneer. Since *Participants* are removed from this structure, *Participants* is initialized anew in every lot.
- *AuctionOn*, in the sense of the *AuctioneerProtocol()*, is what loops through the *Dutch Auction.* Hence, as the *Dutch section* of the protocol ends, *AuctionOn* turns negative: from here on, it either loops through the *English Auction* or ignores it. It won't repeat the *Dutch section.*
- The only condition where the price discussed could be below the *Reserve price* would be during the *Dutch Auction.* As such, this value is only pertinent during it.
- The price increase changes to a fourth of its value to simulate a smaller crescent; yet, the number four's focus is to represent the intention.
- When the *Dutch Auction* terminates, a **Bidder** is assigned as the winner, regardless of the number of messages. Let us call the bidder **DWBidder**. This has a logical reason, and to better understand it, let's reflect on the possible winners on that moment in time, in the code (**line 31**):
  1. There is only one message, and the *English Auction* does not happen. the **DWBidder** wins.
  2. There are several messages, and the *English Auction* proceeds into rounds. The Round Winner overwrites **DWBidder**. No issue.
  3. There are several messages, and the *English Auction* initiates; yet, no one wants to bid higher. Hence, our highest bids were from the last round of the *Dutch Auction.* In this **Conflict case**, the quickest bidder (**DWBidder**) takes the good.
- When the *English Auction*'s bids are organized according to price, if there is a tie, the first one to arrive is the one selected.
- The *approve* messages are, as previously defined, only to inform the **Bidder** he won the **Round**; as such, his **Bid** was the accepted one. Hence, there are two scenarios where this message is sent: If the *Dutch Auction* only had one *proposal*, or to the winner of each *English Round.*
- As implied, the *request* message only proceeds after a **Bidder** had been decided as the **Auction Winner**.
- Although there is no security mechanism regarding the Good and price exchange (as it is not the focus of the assessment), the Good is still sent to the **Bidder**, to simulate the trade. Hence, the **Auctioneer** also has Credit.
- After an **Auctioneer** has sold every item, he can proceed to bid on Auctions. That's the translation of **line 81**.

1.2.3 Bidder's Pseudo-code

Algorithm 2 Pseudo-code representing the Bidder's behaviour.

```
1: BidderProtocol( ):
2: Cravings ← {(Good, HighestPrice, Increase), ...}        /*Auction Goods desired.*/
3: MyId                                                    /*Bidder's Id.*/
4: Credit                                                  /*Bidder's Money.*/
5: Bought ← ∅                                              /*Bought Items.*/
6: Auctions ← ∅                                            /*Keeps track of current Auctions.*/
7: EnglishAuctions ← ∅                                     /*Keeps track of current English Auctions.*/
8: while Credit > 0 do
9:     /*Shout's structure is (MessageType, AuctioneerId, MyId, AuctionId, "..."). */
10:                                                        /*"..." ← arguments dependent on MessageType.*/
11:    Shout ← ReceiveShout(MyId)                          /*Obtain a message sent to the bidder.*/
12:    switch (Shout.MessageType) do
13:        case inform:                                    /*"..." ← "info"*/
14:            if Shout.info ← "Start" then
15:                Auctions ← Auctions ∪ (AuctioneerId, AuctionId)
16:            else if Shout.info ← "English" then         /*Invited to English Auction.*/
17:                EnglishAuctions ← EnglishAuctions ∪ (AuctioneerId, AuctionId)
18:            else if Shout.info ← "over" then            /*Auction is over.*/
19:                remove "(AuctioneerId, AuctionId)" from Auctions
20:                if (AuctioneerId, AuctionId) ∈ EnglishAuctions then
21:                    remove "(AuctioneerId, AuctionId)" from EnglishAuctions
22:                end if
23:            else null                                   /*No process implementing other informations.*/
24:            end if
25:        case call-for-proposal:                         /*"..." ← Good, Price*/
26:            if (AuctioneerId, AuctionId) ∉ Auctions then
27:                SendShout(Not − understood, MyId, AuctioneerId, AuctionId)
28:            else if (Good, HighestPrice, Increase) ∈ Cravings & Price ≤
   HighestPrice then
29:                if (AuctioneerId, AuctionId) ∈ EnglishAuctions then
30:                    if Price + (Price ∗ Increase) > HighestPrice then
31:                        Price ← HighestPrice
32:                    else
33:                        Price ← Price + (Price ∗ Increase)
34:                    end if
35:                end if
36:            end if
37:        case request:                                   /*"..." ← Cost(ofGood)*/
38:            Credit ← Credit − Cost
39:            Bought ← ReceiveGood(MyId)                  /*Receive Good according to Id.*/
40:            remove "Good" from Cravings.
41:        case default:                                   /*Accept, Reject, Propose and Not- understood.*/
42:            null                                        /*No process implementing these Shouts.*/
43: end while
```

1.2.4 Bidder's Discussion

Building on what was regarded during the analysis of the *Auctioneer Protocol*, further assumptions and decisions take the spotlight. These are as follows:

- The variables presented in the beginning of each protocol are presented due to their relevance. The classes concocted will, surely, have more attributes to show (e.g. if the **Auctioneer** can be a **Bidder**, he would have access to the shared and protocol-specific variables, according to the situation).

- *Auction* is indicative of every auction, while *EnglishAuction* is indicative of the auctions inside *Auction* that are being processed as English. This was due to the simplicity this allows: *Auction* to check for active auctions, *EnglishAuction* to check for specifically English Auctions. The first idea was to divide *Dutch Auctions* and *English Auctions* in two different structures; yet, regarding the clauses would be simplified if both could be checked at the same time, the approach taken was believed to be the better option.

- There are three possible *"inform"* messages, equivalent to the ones discussed in the **AUML**: *Start, English* and *over*. The **Auctioneer** does send the winner according to its implementation; yet, although it simulates what would happen in an auction, the *Bidder* is not going to do anything with this information.

- The *Bidder* might also receive *"accept", "reject"*, or, due to late messages from when they were an **Auctioneer**, *"not-understood" and "propose* messages. Obviously, the latter will simply be ignored without much thought. Meanwhile, the former are ignored because there is no processing to do with these messages. The *"request"* message is the one which will determine the **Auction Winner**, according to our implementation, "stealing" the usefulness of the *"accept"* message; alas, *"accept"* is sent to the winner of each round, which is not of big interest regarding the goal of the code.

- As previously commented, the instances of the **Bidder**'s cravings have a different number of arguments from the **Auctioneer**'s belongings, and these will represent different structures. The semantic value of both are very different, as one represents an abstract desire and the other represents a firm object, and the division into two different concepts will help organize our implementation.

2 Implementation

The process of developing the expected software was split into two main phases, due to the very different nature of such. The first phase was to ensure the proper implementation of *Java Remote Method Invocation* (RMI); if the system's design was flawed, communication would be jeopardized, and tackling communication and the protocols at the same time would be deeply prone to errors. After the software's objects were able to communicate remotely just as desired, the protocols would be the following focus.

2.1 Inspiration & Perspective

A big motivation towards the software design was for the classes to be akin to an *Auction*, not only in nomenclatures but also where the processes allowed. The initial approaches to this perspective took root in the first section of this assessment, with, for example, how an **Auctioneer** would become a **Bidder** after his own biddings.

To set up the mood, let the reader imagine the scenery defined for the visualization of the process: The oldest *Auction House* in the world, *Stockholm's Auktionsverk* (Swedish for "Stockholm's Auction House") opened its doors by the early morning. As it is customary, *Auction Goods* are often sold in a time spawn of 20 seconds; so, there is no time to lose! Hence, to ensure time is optimal, the *Auktionsverk*'s organization decided to permit the occurrence of simultaneous auctions and the application of a double-protocol Auction (which we would know as the Dutch protocol and the English protocol). Every invitee would have its credit line confirmed by the *Auktionsverk*, the presence of their account in the *Auktionsverk* confirmed, and two possibilities would now instill: either the invitee would proceed to auctions his items, or, with a bigger interest in affording goods, he'd proceed to take a seat, and amidst every auction, bid in those that caught his interest (although it would be surreal to attend several auctions at the same time in real life, it is brushed off as a detail).

2.2 Communication

To ensure communication, a few decisions were initially made. To start, the platform where the messages between Agents will be exchanged came into realization through the class **AuctionHouse**. The Java RMI "stub" generated by the inherent object will be used as a point of reference for remote objects to communicate between each other. To properly implement the *AuctionHouse*, the *AuctionHouseInterface* was also created, which is the remote interface that defines the *remote methods* realized, and ultimately permits the remote connection of different objects.

Every Agent will be initialized by a single class, which will be called **SessionGenerator**, processing the idea of an *Agent Container*. Each Agent is comprised of three classes:

- *Account* - The class that represents the Agent's Account, and registers the Agent in the *RMI Registry*. It processes the thread-related methods, specifically the *run* method, processed when the thread is initialized, and the *shutdown* method, that terminates the Agent.

- *AccountInterface* - The remote interface that *would* define the remote methods to be implemented. It's easy to notice that the interface is empty. This is because it is only being used to register the Agent's Account in the *RMI Registry*, as there are no methods implemented to be called remotely. The Agents do not have access to each other's RMI registry reference.

- *AuctionPlayer* - The class that represents the *Auctioneer / Bidder*'s behaviour; that is, it implements the pseudo-code previously designed. The Agent's attributes are also stored in this class.

The **Auctioneer** and the **Bidder** will be defined in the same class, called *AuctionPlayer*, to prioritize cohesion and minimize needless coupling. It was also thought to define these as sub-classes of a superclass *invitee*, yet inheritance was believed to turn these two classes disjoint to an undesired degree. As it stands, a person can indulge into any of the two behaviours, according to the simulated clauses.

2.3 AuctionPlayer

As every Agent is initialized as an *AuctionPlayer*, either the **Auctioneer** or **Bidder** Protocol will be initiated, according to certain clauses. To keep it simple, the **Auctioneer** protocol is called if the Agent has any *Belongings* to sell. If he doesn't, the **Bidder** protocol is called instead.

To accommodate every single *AuctionPlayer*, the private attributes for this class are as follows:

- *(AuctionHouseInterface) auktionsverk* - The stub reference that will permit the communication between Agents, by sending messages through the remote *AuctionHouse*.

- *accountId* - The unique Id of the *AuctionPlayer*. This will be the sender and receiver identifier.

- *name* - The name of the *AuctionPlayer*. This is not entirely unique; after all, nothing stops the **Bidders** from having the same name. The *accountId* is the variable that will be used to identify each *AuctionPlayer*, while the *name*'s use is to better understand the actual outcome of the auctions.

- *credit* - The money that the *AuctionPlayer* has. This variable is modified when payments occur, be it by selling an item as an **Auctioneer**, or by buying an item as a **Bidder**.

- *(Map < String, Desire >) cravings* - The desires of an *AuctionPlayer*, which will be taken in account when the **Bidder** protocol is in effect. According to the name of a Good, the *AuctionPlayer* might have a *Desire* towards it, or not. The *Desire* type of Object will be explained in the "Data Structure" section.

- *(List < AuctionFiles >) belongings* - The collection of items the *AuctionPlayer* wants to sell, as a **Auctioneer**. The *AuctionFiles* will also be further explained.

- *(List < Good >) obtained* - The goods the *AuctionPlayers* bought as a *Bidder*.

There is little to comment on the Protocols themselves, as they are very loyal to the pseudo-code. Hence, the explanation of the Data Structures is bound to shine light on the last queries the reader might have. The Data Structure of *Shouts* ("Messages") won't be further investigated as well, as the difference between the implementation and the explanation provided in the *Message Type* and *Message Flow* sections is marginal.

2.4 Data Structures

To comprehend the Data Structures implemented, the first Object type to be explained will be the simplest, which would be the *Good*. From that point onwards, it will be easier to understand the others.

The Data Structure of *Good* couldn't be simpler, as it has a *name*, which is not unique (there can be two people selling two apples); The *owner's name*, which is modified when the good is transacted; and a *finalPrice*, which represents the price at which it was bought.

The *Good* implements the *Serializable* interface, which allows it to be sent through the *AuctionHouse* to a different person. This method is implemented in the *AuctionHouseInterface* for this purpose, and it is similar to the pattern used by the *Shouts* to be sent between *AuctionPlayers*. For each Auction item, there is only one instance of a *Good*. This is important to avoid accidental duplicates, and as such, the *Good* Objects are initialized only once; more specifically, when the *AuctionFile* Data Structure is initialized.

The second Data Structure is the *Desire*, which is formed by the *name of the Good*, the *highest price* the **Bidder** is willing to go, and the *increase rate* the **Bidder** has in mind for the *English Auction*. Each desire is found according to the *Good*'s name, as seen in the designation of *Cravings*.

The last implemented Data Structure is the *AuctionFile*. The first attribute is the *Auction Good*, which is the only place where the *Goods* are initialized. To follow, there are the *initial price* the *Good* is going to be sold for, the *reserve price* and the *increase rate*, which will represent the **Auctioneer**'s increase and decrease.

As it can be perceived, the *Desires* are referent to the **AUctioneer**, while the *AuctionFiles* are referent to the **Bidder**.

2.5 Details & Execution

There are, also, some details worth mentioning:

- To ensure the *auction Id* is unique for each auction, there is a static variable in the *AuctionHouse*, which increases by one with each call of its getter.
- After the auction ends, any input in the console would terminate the auction, and expose some results. More about this in the next section.

In the *AuctionFile* class, there is more besides the Data Structure defined. In it, the reader can also find the static Data Structures that would simulate the "Database" of our system, yet to be populated.

To populate our system, a text file is necessary with the design of our system. This text file is designated as one of the parameters when the *Session Generator* is run from the console, or the *Eclipse IDE* (to be discussed in the next section). The format of such text file is as follows:

```
//Account List (AccountId - Name - Credit)
0,Adam,1000
1,Peter,1000
3,Francis,1000
21,Robert,1000
5,Richard,1000
6,Joseph,1000
23,Stewart,1000
8,God,1000
9,Eve,2000

//Belongings (OwnerId - GoodName - InitialPrice - Reserve - Increase)
0,Tree of Eden,1000,200,0.4

//Desires (InterestedId - GoodName - MaxPrice - Increase)
1,Tree of Eden,350,0.2
21,Tree of Eden,350,0.2
3,Tree of Eden,350,0.2
```

In this example, the *Account List* will be stored in the *AuctionFile*'s data structures, mapping the name of each *AuctionPlayer* to their account Id, just as their credit.

Each belonging and desire will also be inserted into these data structures, mapped according to the account Id it belongs to. Each *AuctionPlayer* will, then, insert in their own Data Structures the information mapped to their account Id.

2.5.1 Using the Software

To run the Software in **Windows and Linux**, the reader must first navigate to the project's folder, where the *security.policy*, the text files and the *agentworld* folder are present. Three command prompts or terminals must be opened and used in this directory, after compiling the folder *agentworld*:

1. The first one will only run the RMI Registry. This is attained through the command **rmiregstry 50000**.
2. The second one will run the *AuctionHouse* class through the command:
   - **java agentworld/AuctionHouse 50000** (Windows)
   - **java agentworld.AuctionHouse 50000** (Linux)
3. The third one will run the *SessionGenerator* class through the command:
   - **java agentworld/SessionManager SessionName 50000 text.txt** (Windows)

- **java agentworld.SessionManager SessionName 50000 text.txt** (Linux)
- The **SessionName** can be any choice of the reader, and the text.txt must present the path to the textfile needed to populate the system.

2.5.2 Results

After the auctions finalize, pressing "Enter" on the *SessionGenerator*'s terminal will reveal information about the auction's outcomes.

3 Experiments & Evaluation

Initially, the proposed implementations were added to the system. These were as follows:

1. To record the number of messages exchanged, the profit of the **Auctioneers** and how much the **Bidders** saved.
2. To enable the creation of experiments (which, at this point, was already functional, regarding the **text file approach** previously documented).
3. To create 3 experiments with at least n **Auctioneers** and n x 10 **Bidders**, reporting the previous metrics.

Due to the design of our system, the implementation of the first step is trivial:

- To implement the number of messages exchanged, the *AuctionHouse* class now contains a static variable *nMessage*, which increments every time a message has been received.
- To implement the profit of the **Auctioneers**, every time an *Auction Good* is sold the difference between the reserve price and the winning bid is summed to the **Auctioneer**'s *profit*. To calculate the total savings, the sum of every **AuctionPlayer**'s *savings* variable is done.
- To implement how much the **Bidders** save, every time an *Auction Good* is bought the difference between the offer and the highest price is summed to the **Bidder**'s *savings*. To calculate the total profit, the sum of every **AuctionPlayer**'s *profit* variable is done.

As the *savings* were being implemented, an interesting situation was found; if there were two auctions occurring for identical items, the **Bidder** with a *Desire* for the item would bid for both, as expected. Yet, if the *Desire* had already been removed due to a successful auction, and it was too late to remove the second auction's bid, at the duplicate item's successful purchase no *Desire* was found to compute how much was saved! Converting this bug into a feature, if the **Bidder** obtains this second item, the price paid will be **deducted** from how much the **Bidder** saved so far. After all, the **Bidder** has no interest in duplicate items, and this punishes such a greedy behaviour.

The metrics defined are added to the results previously referred. To peruse the values of different implementations, press **Enter** after the message requiring such step appears.

3.1 Further evaluation & extensions

To be able to provide an interesting evaluation and easier scalability, some modifications were applied to the system. Two new *features* were implemented, which are the following:

- If the Session's name is defined as *"rand"* when running the *SessionGenerator* class, the *initial Price*, *increase* and *borderline prices* are slightly randomized, inside reasonable constraints.
- If the provided text file does not include *Desires*, the *Session Generator* executes the *populateDesires()* method. This method generates *Desires* for EVERYONE except the **Auctioneer**, with random values within the following intervals:
  1. 100% to 150% of the **Auctioneer**'s reserve price for each *Desire's Highest Price*;
  2. 0.2 to 0.8 for each *Desire's increase*.

The idea is to use these two mechanisms in unison, to be able to run an extensive test without big concerns regarding the fastidious task of designing a very articulate file. Due to how these were designed, when both techniques are applied only the *AuctionFiles* are randomly tweaked, while every *Desire* is randomly generated from root, according to the modified *AuctionFiles*.

To generate 100 **Bidders** (as in the text file sent titled tenHundred), the names can be given little thought, as duplicates do not break the code; yet, the *account Ids* must be unique, and for the **Auctioneer**'s Ids, it's important that they match, obviously, the good they are selling. In the example provided, keep in mind that each **Auctioneer** is selling exclusively one item; the objective of this text file was to comprehend the limitations in performance.

3.2 Performance

To test the performance of our system, several test cases were tested, according to the number of messages and the time elapsed. For the sake of performance, every auctioneer will do one auction, and as such, the number of auctioneers is equal to the latter. The behaviour of the measures were investigated regarding:

- Ten Auctions with 10, 20, 30, 50, 100 and 200 bidders;
- Ten Bidders participating in 10, 20, 30, 50, 100 and 200 auctions;
- 10, 20, 30, 50, 100 and 200 Bidders & Auctions.

In every situation, the randomizing mechanism previously discussed is tested. Each test case was attempted several times, and their average calculated. If the reader desires to test by himself, such test cases are attached to this report.

As the results are perused, it's interesting to note that the growth of the number of messages and the time taken is close to linear (pay close attention to how the x axis grows), except when the number of **Bidders** and **Auctioneers** increase in unison.

This seems to imply that the growth in the number of messages and the time elapsed is *approximating* a linear distribution when only one of these parameters

is increased, while leaning into a exponential distribution (n^2) when both are taken in consideration.

The results can be found at the end of this report.

4 Attachment

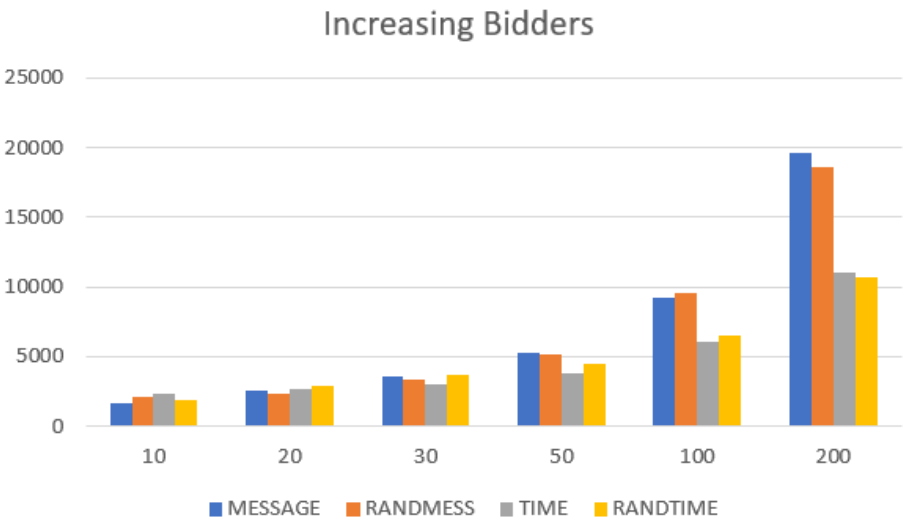


Figure 2: Graph of the increase in number of Bidders.

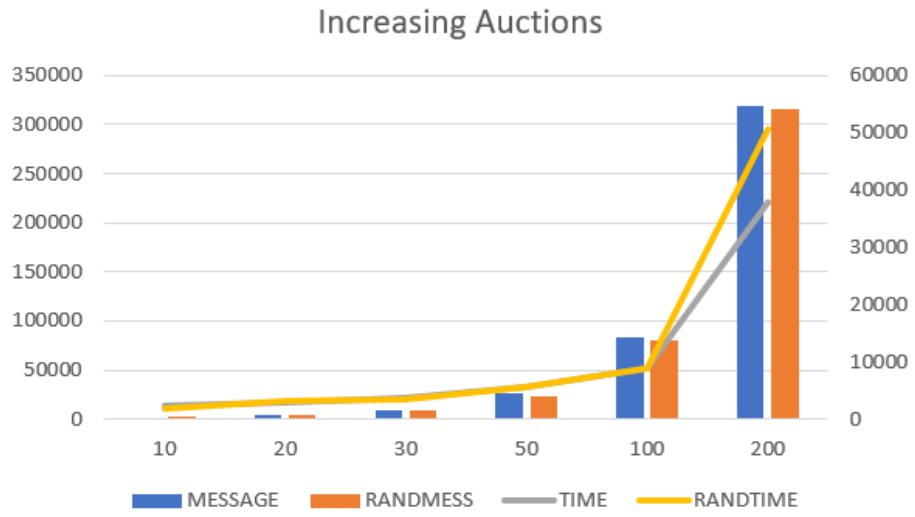


Figure 3: Graph of the increase in number of Auctions.

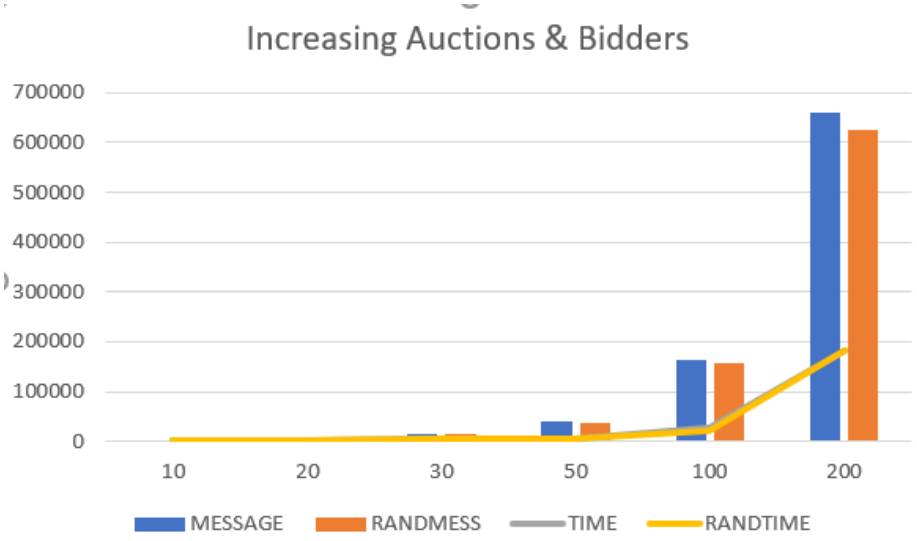


Figure 4: Graph of the increase in number of Auctions & Bidders. The exponential behaviour is the only important property attempting to be illustrated.