Auction-based Agent Negotiation via Programmable Reactive Tuple Spaces

Giacomo Cabri, Letizia Leonardi, Franco Zambonelli
Dipartimento di Scienze dell’Ingegneria – Università di Modena e Reggio Emilia
Via Campi 213/b – 41100 Modena – ITALY
Phone: +39-059-376735 – Fax: +39-059-376799
E-mail: {giacomo.cabri, letizia.leonardi, franco.zambonelli}@unimo.it

Abstract

Auctions are proposed as a distributed negotiation mean, particularly useful in multiagent systems where self-interested agents compete for resources. The Internet is the most known example of environment where agents belonging to different applications exploit the available resources and services. The aim of this paper is to show how auction mechanisms can be easily implemented by using programmable reactive tuple spaces. Tuple spaces are shared repositories of messages that follow the Linda model; the addition of programmable reactivity permits to adapt the behaviour of tuple space to the application-specific requirements. In the implementation of auctions, programmable reactivity is exploited to uncouple the auction mechanisms from selling and bidding policies of the participant agents.

Keyword: Internet, Auctions, Coordination, Tuple Space.

1. Introduction

The Internet world is more and more populated by software agents that perform tasks on behalf of users. The main features that characterise agents are autonomy, proactiveness, reactivity and sociality [JenW98]. While the first three features are related to the development of each single agent application (or of classes of applications) and can be consider together, the fourth one must be discussed at a different level, which can be considered orthogonal to the others. In fact, in the Internet, the social behaviour implies interactions not only with the agents belonging to the same application, but also with agents of different applications, which may have a competitive (self-interested) behaviour to gain the use of resources (whether intended as intangible ones, such as computational resources, data and services, or physical goods in agent mediated E-commerce) [Cle95].

In this context, an interesting negotiation mean among agents is the auction. In an auction there are entities that make resource available and entities that are interested in using such resources. The former ones are usually called sellers, while the latter ones are called bidders. Usually, there is a third entity, called auctioneer, which is the intermediate one that actually performs the negotiation. The price of the resources sold by sellers is not fixed, but it is dynamically determined by the interest of the bidders. The seller can set a reserve price, i.e., a price under which it does not want to sell the resource. Differently from real people, which do not have time or willing
to attend auctions, intelligent agents can spend time to negotiate the desired resources, and the auction mechanisms seem to fit well dynamic and heterogeneous environments. There are several different forms of auction, depending on the number of participants, on the criteria with which the resources are assigned, and so on. We focus on the auctions with one seller and multiple bidders at a time, ruled by the four most common mechanisms: English, Dutch, first-price and Vickery [Ago96].

The aim of this paper is to show how the auction mechanisms can be implemented by exploiting *programmable reactive tuple spaces*. The Linda model defines tuple spaces as repositories of tuples, which are ordered sets of values; the basic operations defined on the tuple spaces permit to put and extract tuples in/from them. Recent proposals show on the one hand that the Linda model well suits heterogeneous scenarios such as the Internet [Cia98] and, on the other hand, that the programmable reactivity of tuple spaces permits to uncouple algorithmic issues from interaction issues, leading both to an easier programming style and to a clearer separation of concerns [CabLZ99, DenNO98]. This paper shows not only that auctions can be implemented on a tuple space, exploiting tuples to let agent locally interact, but also that programmable reactivity makes such implementation easy, and permits to uncouple the definition of agents from the management of the auctions. This permits average programmers to focus on selling and buying policies, disregarding the implementation of auction mechanisms; however, programmable tuple spaces allow expert programmers to install specific application-depending mechanisms.

The paper is organised as follows. Section 2 introduces agent coordination models and, in particular, presents the MARS architecture. Section 3 shows how the four kinds of auction can be implemented by using MARS. Section 4 reports the conclusions and the open issues.

### 2. MARS

This section starts with some considerations about the coordination models for agents, and then presents MARS [MOON97], a programmable coordination architecture based on the Linda model. MARS was mainly conceived for mobile agents, i.e., for agents that can actively change their execution environment during their life; however, the considerations and the results of this paper are not related to the capability of mobility and can be applied to general multiagent Internet systems.

#### 2.1 Background

In a previous work [CabLZ99], we discussed in detail about the coordination models that can be adopted in Internet applications, by arguing that the Linda model suits well the requirements of such applications. In the Linda coordination model [AhuCG86], interactions occur via shared data repositories (*tuple spaces*) whose data is organised in terms of ordered sets of typed fields (*tuples*), e.g., *(int 5, char ‘c’, float 3.14)*. Interactions occur via three operations on the tuple space: one output operation is provided to store a tuple; two input operations are provided to retrieve a tuple, one of which also extracts it. Data retrieval relies on a *pattern-matching* (associative) mechanism: a tuple with possibly some non-defined values (*template*), e.g., *(int?, char ‘c’, float?)*, is provided with the input operations; a tuple is retrieved from a space if it corresponds to (*matches with*) the template [GelC92].
Linda has been widely recognised as a powerful and general coordination model for parallel and distributed applications. This is mainly due to the fact that it enforces both spatially and temporally uncoupled interactions: interactions can occur via one tuple space without needing to know who the interacting partners are, where they are, and when they have been (or will be) involved in the interaction. In addition, since interactions are based on associative access to tuples, the Linda model permits to acquire information also on the basis of partial knowledge.

To implement agent coordination in Internet applications, a Linda tuple space can be associated with each node of the network. The node administrator can publish on the tuple space data and services she/he wants to make available. Agents access the tuple space to retrieve data and trigger service activities in an associative way. Tuple spaces can also be effectively exploited for inter-agent coordination: agents can write tuples representing messages in a tuple space, to be read or consumed by other agents at a proper time, thus avoiding the need for mutual localisation and identification of the interacting agents.

2.2 MARS Programmable Reactivity

Despite its advantages, the static data-oriented interaction Linda model is too restrictive and static to answer the needs of current dynamic and distributed Internet applications. The main focus of this model is on associative data access, and the pattern-matching mechanism exploited to retrieve information, embedded in the tuple space, triggers the same mechanism whatever the type of access to the tuple space. However, the Internet is not simply a web of data, but of services and other agents too, where every interaction usually occurs in an uncertain and untrusted scenario. In this context, the basic Linda model presents the following drawbacks:

- it is unable to provide flexible solutions to manage services in a simple and effective way;
- it does not provide any way to monitor and control the interactions occurring via a tuple space;
- it forces complex and odd solutions when agents have to be involved in complex interaction protocols;

In order to maintain the advantages and the simplicity of the Linda model, and in the effort of enhancing it to overcome the above drawbacks, we exploit the concept of *programmable reactive tuple spaces*. A programmable reactive tuple space has the capability of embodying computational capacity within the tuple space itself (*programmable* property), assuming specific behaviours in response to access events (*reactive* property) [DenNO98]. Then, the tuple space is no longer a mere tuple repository with a built-in and stateless associative mechanism, as in Linda. Instead, it can also have its own state and can be programmed to react with specific actions to specific accesses to it performed by agents. Reactions can access the tuple space, change its content and influence the semantics of the accesses to the tuple space.

A programmable tuple space model overcomes the above-identified drawbacks of the basic Linda model:

- services can be accessed by agents as data in a uniform way, by using the normal access operations together with the programmable reactive capabilities;
- by associating a specific behaviour, one can monitor all access events and also associate specific control policies for handling them;
• a tuple space enables to specify into the tuple space even complex interaction protocols, thus freeing the agents from the duty of explicitly managing and controlling their interactions with other agents.

With regard to the latter point, a programmable tuple space allows the specification of inter-agent coordination rules in terms of reactions, thus achieving a clean separation of concerns between algorithmic and coordination issues [GelC92]. Agents are in charge of embodying the algorithms to solve the problems; reactions represent the application-specific coordination rules.

2.3 The MARS Architecture

MARS (Mobile Agent Reactive Spaces) [CabLZ98] implements a Linda-like coordination architecture based on programmable tuple spaces to be used by Java-based mobile agent applications. The MARS system is conceived for the coordination of Java-based agent applications. It is assumed that each node of the network hosts an agent server in charge of executing autonomous Java agents, for example Aglets [LanO98]. The MARS tuple space is only loosely coupled to the agent server and can be associated with different systems and implementations. The agent server keeps a reference to a local tuple space: as an agent is created or arrives on a node (step a of Figure 1), the local server provides it this reference (step b). As an agent is bound to the local tuple space, it can access it for reading, extracting and putting in tuples (step c in Figure 1). Then, an agent associatively retrieves from the tuple space primitive data items and references to objects that represent execution environment resources.

![Figure 1. The MARS Architecture (smiles represent agents)](image)

MARS implements a **programmable reactive tuple space** model in which the effects of the operations on the tuple spaces can be dynamically modified. Agents always access the tuple spaces with the same basic set of Linda-like operations. Specific reactions can be associated to the accesses to the tuple spaces made by agents (step d in Figure 1). A meta-level tuple space is introduced in each node to associatively manage reactions.

2.4 The MARS Interface

Each tuple space is a Java object, an instance of the Space class, which implements the interface with which agents can access the tuple space. The MARS interface (see Figure 2) extends
that of the SUN JavaSpaces specification [JS98], that is likely to become the de facto standard tuple space interface for Java.

```java
public interface MARS extends JavaSpace
{
    // method interface inherited from JavaSpace; operations can belong to a transaction txn
    // Lease write(Entry e, Transaction tx, long lease);
    // put the tuple e into the space, where remains for a time lease
    // Entry read(Entry tmp, Transaction tx, long timeout);
    // read a tuple matching with tmp from the space, waiting for timeout before returning null
    // Entry take(Entry tmp, Transaction tx, long timeout);
    // extract a tuple matching with tmp from the space, waiting for timeout before returning null

    // methods added by MARS
    Vector readAll(Entry tmp, Transaction tx, long timeout);
    // read all tuples matching with tmp from the space, waiting for timeout before returning null
    Vector takeAll(Entry tmp, Transaction tx, long timeout);
    // extract all tuples matching with tmp from the space, waiting for timeout before returning null
}
```

**Figure 2. The MARS interface**

MARS tuples are Java objects that represent ordered sets of typed fields. Tuple classes must implement a specific interface required for tuple management and named Entry in JavaSpaces, or, in a simpler way, can sub-class from the so called AbstractEntry class, which provides for a basic implementation the Entry interface. The specific tuple fields are defined as instance variables of the tuple classes. Each field of the tuple refers to an object that can also represent primitive data (wrapper objects in the Java terminology). The JavaSpace interface defines three operations to access the tuple space, basically with the same semantics of the Linda operations:

- **write**, to put a tuple, supplied as parameter, in the space;
- **read**, to retrieve a tuple from the space, on the basis of a request tuple supplied as a parameter and to be used as a pattern for the matching mechanism;
- **take**, which works as the read operation but extracts the matching tuple from the space.

Moreover, MARS adds two operations, called readAll and takeAll, which permit to read/extract all tuples matching with the given template. The template tuple supplied by reading and taking operations can have both actual (defined) and formal (null) values. Because in MARS (as in JavaSpaces) tuples are objects and because the elements of a tuple can be non-primitive objects, the matching rules must take into account the presence of objects: objects match if their corresponding fields have the same values and so on in a recursive way, if fields are objects again.

A meta-level tuple in the form (Reaction_object, T, O, I) associates the reaction implemented by Reaction_object with the event of the operation O performed on the tuple T by the agent with identity I. Writing a meta-level tuple in the meta-level tuple space means installing the corresponding reaction, while taking a meta-level tuple means uninstalling the corresponding reaction. Readings on the meta-level tuple space are performed by the system to search for a matching reaction when an operation occurs on the base-level tuple space. Since T can be a template, a single meta-level tuple can be used to associate a reaction with the events related to all the tuples matching with T.
3. Implementation of Auctions

Auctions are a general negotiation mean that can be implemented in different ways.

The first choice could be the traditional message-passing model, possibly integrated with the Web [WWW98]. In this case, a server agent can implement the auctioneer, which is in charge of dealing with one participant agent. First of all, it has to accept the incoming requests of initiating an auction from the seller agent. Then, such auction must be published in some way to advise all possibly interested agents. Further, the server must keep track of all interested agents (which has to register themselves at the server), manage the bids and acknowledge all participants of any change related to the current higher bid.

![Figure 3. An auction implemented via a programmable reactive tuple space: a seller agent puts a good on sale (a), the bidder agents put their bids (b), a reaction decides the winner (c)](image)

The use of a non-programmable tuple space can help to uncouple the management of the auction: the auctioneer agent does not have to keep track of all participants agents, because all messages are exchanged via the tuple space in an anonymous way. Nevertheless, by using the fixed pattern matching of Linda, the auctioneer or the seller are in charge of embodying the code needed to manage the auction.

Programmable reactive tuple spaces can be used to make the programming easier. In general, an auction implemented via a programmable tuple space works as follows. The seller agent writes a tuple that contains information about the resource it is going to sell (step a in Figure 3); this writing triggers a reaction that acts as the auctioneer, i.e. it is in charge of managing the auction. Once bidder agents have read such tuple, they can bid a price (or more than one, depending on the auction type) to buy the resource on sale (step b in Figure 3); the previous-triggered reaction monitors the bids from the agents. When the auction is over, the auctioneer via the reaction decides the winner agent and creates a tuple to inform all the participants about it (step c in Figure 3).

Our auction implementation described in the rest of the paper makes use of MARS security capabilities to authorise only correct readings, takings and writings of tuples.

In the following we describe some kinds of tuples, implemented by Java classes, representing general entities that are useful in the auction management. The first class is called AuctionEntry and represents the announcement of an auction (Figure 4). It is written in the tuple space by the seller agent to advise other agents that a given resource is sold by an auction. The interesting fields stored in the tuple are:
• the *description* of the resource sold, which uniquely identify the resource/good/service on sale (in this case a string is used, but it could be a more complex information, such as an XML document);
• the *deadline* of the auction, i.e. the time after when no bids are accepted;
• a reference to the *seller agent*, which wants to sell the resource;
• a *reserve price*, under which the seller agent does not want to sell the resource;
• the *type* of the auction.

```java
class AuctionEntry extends AbstractEntry { // AbstractEntry: root of the tuple hierarchy
    public String description; // description on the resource
    public Date deadline; // deadline of the auction
    public Seller seller; // a reference to the seller agent
    public Integer reserve; // the lowest admissible price
    public String type; // the type of the auction

    // constructor of the tuple. The order of the parameters defines the tuple field order
    public AuctionEntry(String descr, Seller seller, Date deadline, int reserve, String type)
    { this.description = descr;
      this.deadline = new Date(deadline);
      this.seller = seller;
      this.reserve = new Integer(reserve);
      this.type = type;
    }
}
```

**Figure 4. The AuctionEntry class**

This class can be subclassed to define tuple classes that represent information of a given kind of auction, adding more fields if the basic ones are not enough to represent all useful information.

```java
class BidEntry extends AbstractEntry { // AbstractEntry: root of the tuple hierarchy
    public String description; // description on the resource
    public Integer price; // the bid price
    public Bidder bidder; // the bidder agent
    public String type; // the type of the auction

    // constructor of the tuple. The order of the parameters defines the tuple field order
    public BidEntry(String descr, int price, Bidder bidder, String type)
    { this.description = new String(descr);
      this.price = new Integer(price);
      this.bidder = bidder;
      this.type = type; }
}
```

**Figure 5. The BidEntry class**

The second tuple class represents a bid for the auction (Figure 5). Bidder agents use it to offer an amount of money in order to buy the given resource. The fields stored in these tuples are:
• the *description* of the resource for which the agent bids;
• the *price* the agent bids;
• a reference to the *bidder agent*, which wants to buy the resource;
• the *type* of the auction.
The WinEntry class reported in Figure 6 is used to publish tuples that advise all the participant agents about the winner of an auction.

```java
class WinEntry extends AbstractEntry { // AbstractEntry: root of the tuple hierarchy
    // tuple fields
    public String description; // description on the resource
    public Seller seller; // the bid price
    public Bidder bidder; // the bidder agent
    public Integer price; // the final price

    // constructor of the tuple. The order of the parameters defines the tuple field order
    public AuctionEntry(String descr, Seller seller, Bidder bidder, int price)
    {
        this.description = new String(descr); this.seller = seller;
        this.bidder = bidder; this.price = new Integer(price);
    }
}
```

**Figure 6. The WinEntry class**

The previous tuples use references to seller and bidder agents, by using fields whose types are the interfaces Seller and Bidder; they contain specific useful methods to permit mutual identification after the auction. The Figure 7 presents a fragment of code of an agent that wants to sell a space of 5 Mbytes on a local HD via a first-price auction. The seller agent simple publishes its announcement by writing an AuctionEntry tuple in the space (s.write statement) and then waits for a WinEntry tuple describing the winner of the auction (s.read statement). It does not care about the management of the auction, because it is delegated to the reaction, which becomes the auctioneer, triggered when the AuctionEntry tuple is written. The meta-level tuple that associates a first-price auction reaction to the writing of an AuctionEntry tuple is the following: (FP Auction_Obj, AuctionEntry_tuple, “write”, null), where EnglishAuction_Obj is an instance of the class EnglishAuction presented later and AuctionEntry_tuple is an instance of AuctionEntry and has the following values: (null, null, null, null, “first-price”). As shown in the Figure 7, the code of the seller agent is very simple, because it does not contain all the instructions needed to manage the auction.

```java
MARS s; // reference to the MARS interface to access the reactive tuple space
...
// sell 5 Mb of HD space, within 10 seconds, a reserve price of 10 units, by a first-price auction
AuctionEntry tuple = new AuctionEntry(“5Mb@HD”, this, new Date(System.currentTimeMillis() + 10 * 1000), 10, “first-price”);
// this writing triggers the appropriate reaction that manages the auction
s.write(tuple, new Transaction(null), 600);
// wait for a response
WinEntry winAdv = (WinEntry)s.read(new WinEntry(“5Mb@HD”, this, null, null), new Transaction(null), 660);
// if there is a winner
if (winAdv.bidder != null)
    // interact with it …
```

**Figure 7. Fragment of the code of a seller agent exploiting a programmable tuple space**

Figure 8 shows a fragment of the code of a bidder agent that wants to buy the 5 Mbytes of HD space. It searches for a tuple that publishes the resource it wants to acquire; if an auction is found, the agent puts its bid in the form of tuple and waits for the result.
MARS s; // reference to the MARS interface to access the reactive tuple space

// want to buy 5 Mb of HD space by a first-price auction
AuctionEntry tuple = new AuctionEntry("5Mb@HD", null, null, null, “first-price”);
if (s.read(tuple, new Transaction(null), NO_WAIT) != null)
{
    // put a bid of 15 units
    BidEntry bid = new BidEntry("5Mb@HD", 15, this, “first-price”);
    WinEntry winAdv = (WinEntry)s.read(new WinEntry("5Mb@HD", null, null, null), new Transaction(null), 0);
    // if I am the winner
    if (winAdv.bidder == this)
        // interact with the seller
...}

Figure 8. Fragment of the code of a bidder agent exploiting a programmable tuple space

In the case shown in Figure 8 the bidder chooses a given kind of auction. In a more general case, the bidder agent can search only for a given description and it knows the kind of auction when it has retrieved the tuple; then it deals with the auction in the appropriate way.

If a seller agent wants to use another kind of auction, it has to change the last field in the AuctionEntry tuple, and specifies the desired auction. If it wants to use a new kind of auction (of which it has the code in the form of reaction), it has to install the reaction via a meta-level tuple such as (CustomAuction_Obj, AuctionEntry_tuple, “write”, null), where again AuctionEntry_tuple is an instance of AuctionEntry and has the following values: (null, null, null, null, “Custom”). Then it has to specify “Custom” in the last field of the AuctionEntry tuple written in the tuple space to publish the auction.

3.1 English Auctions

The English auction is an out-cry auction, i.e., all the bids are public. The seller starts with an entry price and asks the bidders if there is a bid with a price higher than the initial one. Each participant can bid a price higher from the previous one. When there is no bid for a given interval of time, the last bidder wins the auction and gains the resource.

class CurrentBidEntry extends BidEntry
{
    // tuple fields

    // constructor of the tuple. The order of the parameters defines the tuple field order
    public CurrentBidEntry(String descr, int price; Bidder bidder, String type)
    {
        super(descr, price, bidder, type);
    }

Figure 9. The CurrentBidEntry class

The AuctionEntry shown in Figure 4 permits to publish the sale of something by an English auction, specifying the Type field with the string “English”. At the same way, the BidEntry shown in Figure 5 is used to represent a bid in the auction, specifying the Type field with the string “English”. A class deriving from BidEntry, the CurrentBidEntry shown in Figure 9, is used to publish the current bid made by a bidder. In this way, all participant agents can know which the current bid is.
class EnglishAuction extends Reactivity
{
    private String description;
    private Date deadline;
    private Seller seller;
    private int reserve;

    public Entry reaction(Space s, Entry Fe, Operation Op, Identity Id)
    // this method is executed when the reaction is triggered and initializes the parameters
    {
        description = (AuctionEntry)Fe.description;
        deadline = new Date((AuctionEntry)Fe.deadline);
        seller = (AuctionEntry)Fe.seller;
        reserve = (AuctionEntry)Fe.reserve.intValue();
    }

    public void run()
    // this method is executed in a separate thread when the reaction is triggered
    {
        int timeStep = 300;       // 5 minutes with no bid to assign the resource
        CurrentBidEntry currentBid;
        BidEntry lastBid, bidTemplate;

        currentBid = new CurrentBidEntry(description, null, null, "English");
        bidTemplate = new BidEntry(description, null, null, "English");

        // retrieve the last bid
        while (new Date().before(deadline) &&
            ((lastBid = (BidEntry)s.take(bidTemplate, new Transaction(null), timeStep)) != null))
        {
            // delete the current bid tuple
            s.take(currentBid, new Transaction(null), NO_WAIT);
            if (lastBid.price.intValue() > currentBid.price.intValue())
            {
                // update the current bid on the basis of the last bid
                currentBid.price = lastBid.price;
                currentBid.bidder = lastBid.bidder;
            }
            // publish the current bid tuple in the space
            s.write(currentBid, new Transaction(null), timeStep);
        }

        // the seller and the winner are notified by a tuple
        WinEntry winAdv = new WinEntry(description, seller, null, 0);
        if (currentBid.bidder != null)
        {
            winAdv.bidder = currentBid.bidder; winAdv.price = currentBid.price;
            s.write(winAdv, new Transaction(null), 1000);
            // delete old tuple
            s.take(currentBid, new Transaction(null), NO_WAIT);
        }
    }
}

Figure 10. The EnglishAuction reaction class

The auctioneer is implemented by the auction reaction (the method run shown in Figure 10),
which consists of a while cycle, whose conditions are that the auction is not over and there is a new
bid within a given time (5 minutes in the example). The price at which the auction starts is the
reserve price, i.e., the lowest price for the seller. The cycle body controls whether the new bid
(lastBid) is higher than the previous one (currentBid); in positive case, the current-bid tuple is updated
and written in the space. In this way, agents attending the auction are informed about the new bid.
When there are no bids for a prefixed time (5 minutes in the example) or the auction is over, a tuple with the reference of the winner (or null if none bids) is written to notify all the participants.

### 3.2 Dutch Auction

In the Dutch auction there is only one bid for auction. In fact, the seller offers the resource at a given (high) price; if no bidder accepts the price, the seller decreases the price and asks if someone accepts the new offer. The seller keeps on decreasing the price until a bidder signals the willing of buy the resource at the current price. In Dutch auctions bids are public.

The class that describes a Dutch auction tuple derives from the AuctionEntry one and adds the `startPrice` field that define the price at which the auction starts (see Figure 11). In fact, differently from the previous case, the starting price is not the reserve price, but it is higher.

```java
class DutchAuctionEntry extends AuctionEntry
{
  // tuple fields
  public Integer startPrice; // the price from which the auction starts

  // constructor of the tuple. The order of the parameters defines the tuple field order
  public DutchAuctionEntry(String descr, Date deadline, Seller seller, int reserve, int startPrice)
  {
    super(descr, deadline, seller, reserve, "Dutch");
    this.startPrice = new Integer(startPrice);
  }
}
```

**Figure 11. The DutchAuctionEntry class**

The class that implements a bid in a Dutch auction is the BidEntry already reported in Figure 5. The Type field is set to “Dutch” to specify the kind of auction it belongs. Also, as in the English auction, for the management of the Dutch auction is also used the CurrentBidEntry class, shown in Figure 9; in this case, it represents the current bid of the auctioneer.

The auctioneer of the Dutch auction is implemented by the reaction class shown in Figure 12. The method run consists of a while cycle that is repeated until a bid is found. If there are no bids for a given time (1 minute in the example), the price is decreased of a given amount (5 units in the example) and it is offered to the participants. When an agent makes a bid, the cycle terminates and the agent gains the good on sale; a tuple with the reference of the winner (or null if none bids) is written to notify all the participants.

```java
class DutchAuction extends Reactivity
{
  private String description;
  private Date deadline;
  private Seller seller;
  private int reserve;
  private int startPrice;

  public Entry reaction(Space s, Entry Fe, Operation Op, Identity Id)
  // this method is executed when the reaction is triggered and initializes the parameters
  {
    description = (DutchAuctionEntry)Fe.description;
    deadline = new Date((DutchAuctionEntry)Fe.deadline);
    seller = (DutchAuctionEntry)Fe.seller;
    reserve = (DutchAuctionEntry)Fe.reserve.intValue();
    startPrice = (DutchAuctionEntry)Fe.startPrice.intValue();
  }
}
```
```java
public void run()
// this method is executed in a separate thread when the reaction is triggered
{
    int timeStep = 60; // after 1 minute the price is decreased
    int priceStep = 5; // the price is decreased of priceStep each time
    BidEntry lastBid, bidTemplate;
    CurrentBidEntry currentBid;

    bidTemplate = new BidEntry(description, null, null, "Dutch");
    // the current bid starts from the given price
    currentBid = new CurrentBidEntry(description, null, startPrice, "Dutch");
    s.write(currentBid, new Transaction(null), deadline);

    // search for a bid
    while ((lastBid = (BidEntry)s.take(bidTemplate, new Transaction(null), timeStep)) == null)
    {
        // timeStep is over and there are no bids
        // update the current bid with a lower price
        s.take(currentBid, new Transaction(null), NO_WAIT);
        currentBid.price = new Integer(currentBid.price.intValue() - priceStep);
        if (currentBid.price.intValue() < reserve) break; // the price is too low: no winner
        // publish the current bid tuple in the space
        s.write(currentBid, new Transaction(null), timeStep);
    }

    // the seller and the winner are notified by a tuple
    WinEntry winAdv = new new WinEntry(description, seller, null, 0);
    if (currentBid.price.intValue() >= reserve)
    {
        winAdv.bidder = lastBid.bidder; winAdv.price = currentBid.price;
        // publish the current bid tuple in the space
        s.write(winAdv, new Transaction(null), 1000);
        // delete old tuples
        s.take(currentBid, new Transaction(null), NO_WAIT);
    }
}
```

**Figure 12. The DutchAuction reaction class**

### 3.3 First-Price Sealed-Bid Auction

The first-price auction differs from the previous two because the bids are not public, but private. Each bidder does not know the bids of the other participants. It is important that all bids are kept secret until the end of the auction. At the given time, the auctioneer opens all the bids and decides the winner, which is the bidder of the highest bid.

To publish the announcement of a first-price auction, a tuple of kind AuctionEntry (see Figure 4) can be used, specifying the kind of auction via the string “first-price” assigned to the Type field. As usual, the bid is represented by a tuple of the BidEntry class (see Figure 5).

The class that implements a first-price auctioneer is shown in Figure 13. It works quite differently with regard to the previous ones. In this case, the reaction “sleeps” until the deadline of the auction, while the agents put their secret bids into the tuple space. Then, it wakes up and retrieves all the bid tuples. The bidder with the highest price is the winner and all the participants are informed by an appropriate tuple written in the space.
class FPAuction extends Reactivity
{
    private String description;
    private Date deadline;
    private Seller seller;
    private int reserve;

    public Entry reaction(Space s, Entry Fe, Operation Op, Identity Id)
    // this method is executed when the reaction is triggered and initializes the parameters
    {
        description = (AuctionEntry)Fe.description;
        deadline = new Date((AuctionEntry)Fe.deadline);
        seller = (AuctionEntry)Fe.seller;
        reserve = (AuctionEntry)Fe.reserve.intValue();
    }

    public void run()
    // this method is executed in a separate thread when the reaction is triggered
    {
        try  // sleep until the deadline
        { sleep(deadline.getTime() – new Date().getTime())
        catch (InterruptedException e) {}}

        BidEntry bidTemplate = new BidEntry(description, null, null, “first-price”);
        BidEntry bids[];  // retrieve all bids from the space
        bids = (BidEntry[])s.takeAll(bidTemplate, new Transaction(null), NO_WAIT);

        int MaxPrice = 0;
        Bidder winner = null;
        if (bids != null)  // if there are bids
        {
            for (int i = 0; i < bids.length; i++)  // search for the highest bid
            {
                if (bids[i].price.intValue() > MaxPrice)
                { MaxPrice = bids[i].price.intValue();
                  winner = bids[i].bidder;
                }
            }
            if (MaxPrice < reserve) // if the highest bid is less than the lowest admissible
            { winner = null; }  // no one wins the auction
        }

        // the seller and the winner are notified by a tuple
        WinEntry winAdv = new new WinEntry(description, seller, winner, MaxPrice);
        s.write(winAdv, new Transaction(null), 1000);
    }
}

Figure 13. The FPAuction reaction class

3.4 Vickery (Uniform Second Price) Auction

The Vickery auction is quite similar to the first-price one described in the previous subsection. Therefore the secret bids are opened by the auctioneer at a given time to decide the winner. The only difference is that the bidder with the highest bid wins the auction, but she/he pays the second highest price [Vic61]. For example, if the bids are 5, 6, 9 and 10, the agent who bids 10 wins, but it pays only 9. With respect to the first-price auction, in the Vickery auction the bidders
are led to offer prices that reflect their true interest in the resources on sale. In this kind of auction, it is very important to keep the bids secret, more than in the previous one.

The tuple that describes a Vickery auction is the AuctionEntry (see Figure 4) with the Type field set to the string “vickery”. The related bids are similar to the previous ones, with the only difference that they define the string “vickery” as the type of auction (see Figure 5).

The work of the reaction class (shown in Figure 14) is very similar to the previous auctioneer’s one. The only difference is that the second price must be recorded in a variable, and it has to be published as the price the winner must pay.

```java
class VickeryAuction extends Reactivity
{
    private String description;
    private Date deadline;
    private Seller seller;
    private int reserve;

    public Entry reaction(Space s, Entry Fe, Operation Op, Identity Id)
    // this method is executed when the reaction is triggered and initializes the parameters
    { //... same as first-price }

    public void run()
    // this method is executed in a separate thread when the reaction is triggered
    {
        ...  // same as first-price except the for cycle that takes into account the second price:
        int SecondPrice = 0;
        for (int i = 0; i < bids.length; i++) // search for the highest and the second highest bid
        {
            if (bids[i].price > MaxPrice)
            {
                SecondPrice = MaxPrice;
                MaxPrice = bids[i].price;
                winner = bids[i].bidder;
            }
            else if (bids[i].price > SecondPrice)
                SecondPrice = bids[i].price;
        }
        // the notification is the same of the first-price except for the price
        WinEntry winAdv = new new WinEntry(description, seller, winner, SecondPrice);
        s.write(winAdv, new Transaction(null), 1000);
    }
}
```

Figure 14. The VickeryAuction reaction class

4. Conclusions

This paper has shown the power of the programmable reactive tuple space model in implementing higher-level interaction mechanisms such as the auctions. This model brings several advantages in the area of agent coordination. First, the programmable reactivity permits to uncouple the algorithmic issues from the coordination issues, embodying the first ones in the code of agents, while the second ones are implemented via reactions. Secondly, the presented code shows that the implementation of auctions is made easy by the programmable model embodied in MARS.
Agent programmers can rely on already-coded and tested mechanisms made available from the site that hosts auctions, without the need of coding them into the agents. Moreover, application-specific reactions (which act only on the tuples of the application that installs them) can be installed in the auction tuple space if the programmer or the administrator decide to change the standard policies. This paper has presented the code to implement the basic mechanisms of auctions; it can be extended to take into consideration peculiar situations that can occur in auctions.

A research direction related to resource allocation is the management of payments in open and wide networks such as the Internet. The virtual money is a very attractive idea, but there are several security issues that are to be faced before a wide scale exploitation of payment mean. Even if the auction mechanisms are not strictly related to the means of payment, the integration of the two issues will produce systems that are more complete and usable in the real life. More in general, the integration of auction mechanisms in e-commerce systems [San99] will lead to a general market system where people (or agents) can sell and buy goods and services.

5. References


