# Teaching Distributed Programming Concepts using a Java and Logo-based Framework

Thanaporn Chochai[1]
Telecom. and Network Research and Development Division
NECTEC
Rajtawe, Bangkok 10400, Thailand
rung@ttl.nectec.or.th

Andrew Davison[1]
Dept. of Computer Engineering
Prince of Songkla University
Hat Yai, Songkhla 90112, Thailand
dandrew@ratree.psu.ac.th

## Abstract

*This work describes a Java and Logo-based framework, which facilitates an understanding of distributed programming in an active learning setting.*

*The framework, called MultiWorldLogo, simulates multiple separated grids (worlds) with interacting entities, teaches the elements of message passing, and introduces the mobile agent paradigm. It includes system-configurable unreliability (i.e. messages may disappear or arrive out of order), and two forms of mailbox. The visualization of these complex ideas is fully realized; animation is used to model the dynamic notions of entity movement, migration, and message passing.*

## Keywords/phrases

Interactive learning environment, multimedia and visualization in classroom teaching, logo-based framework, Java, understanding distributed programming.

## 1 Introduction

The vast range of distribution scenarios, combined with the spectrum of choices in cooperation and communication, make distributed programming one of the hardest practical areas of computing [1]. Although many languages and systems can be used for explaining distributed ideas, they are frequently too complex and/or too detailed for novices (e.g. RMI and CORBA).

Active learning is the notion that a student learns best by taking part in the learning process rather than passively receiving instruction [4]. The Logo programming language has a long history as an excellent active learning tool. Logo encourages the development of problem-solving skills without being too rich a language.

Our Java-Logo framework supports the teaching of distributed programming ideas within the context of a simulated world of 'grids' inhabited by mobile entities, mailboxes, and messages. Communication can be configured to be unreliable (i.e. messages disappear, arrive out of order). Entities move around a grid, and can migrate between grids. The framework exists as a set of Java classes which support these distribution concepts in a Logo-like style.

## 2 Logo Background

StarLogo is a massively parallel programming language for developing decentralized systems (i.e. those without a central controlling unit) [5]. It supports 1000's of concurrently executing turtles. Each one follows a set of simple rules which lead to an emergent pattern of behavior for the whole system. The single grid is divided into small squares called patches. Each patch has the same computational attributes as a turtle, but cannot move.

StarLogo was the inspiration for the Java-based framework developed in [6]. Its main contribution is the idea of representing turtles, patches, and the grid, as classes. This permits

[1] This work was started while the authors were members of the CSIM Program at the Asian Institute of Technology (AIT), Bangkok, Thailand. Its generosity is kindly appreciated

applications to subclass the basic framework, which facilitates prototyping.

The aims of both systems are to teach concurrent programming notions, such as agency decomposition and synchronization. Our work extends these ideas to a distributed setting of multiple grids.

Orespics-PL contains traditional imperative statement primitives, such as `repeat`, `while`, `if`, in addition to the usual Logo commands [2,3]. Moreover, the language offers a wide range of primitives for synchronous and asynchronous communication between turtles residing on a single grid, including broadcasting and multicasting. This means that both centralized and decentralized models can be represented. However, the single grid basis of the system prevents issues related to unreliable communication between grids from being explored.

## 3 Framework Details

Our Java-Logo framework, called MultiWorldLogo, has six main components used for programming:

*Grid*. Turtles can move about (and between) grids. Multiple grids offer an analogy with physically separated distributed systems. Each grid has a unique ID, which allows it to be addressed easily. A grid consists of StarLogo-style patches.

*Turtle and Patch*. The Turtle and Patch classes support basic methods (e.g. turtle movement).Through inheritance, new types of turtles and patches can be defined in a straightforward manner. The turtle and patch classes inherit from the PatchTurtle class which holds common functionality, such as position information. Patches and turtles can be supposed to perform their activities in parallel. However, this parallelism is actually modeled as a sequence of discrete steps: a time-stepping algorithm updates all the entities in each time interval.

*Message*. A message includes the sender and receiver IDs: a message can only be addressed to one receiver; there is currently no broadcasting or multicasting. In an unreliable communications environment, messages may not arrive, or arrive in a different order from which they were sent. This introduces students to important problems inherent in distributed models. Messages are sent to mailboxes.

*Mailbox*. The advantage of mailboxes is support for asynchronous, decoupled communication: a sender need not wait for a receiver before sending a message. The framework contains turtle and patch mailboxes. A turtle mailbox is automatically created for each turtle; only the turtle owner can read its messages. A patch mailbox may be placed on a patch, and so can be shared by turtles. However, messages can only be read by their designated receiver.

*Name server*. When a turtle wishes to communicate with turtles on other grids, or to migrate to a different grid, it needs to know the destination grid ID. The name server offers a service for finding grid references.

Our framework is partly based on the one in [6]. The main area of commonality is the use of a time-stepping algorithm to update the turtles and patches in lock-step. This is implemented by a MainLoop object calling the `update()` method in each turtle and patch at every time step. When every entity has been updated the display is redrawn. The practical advantage of this approach is the simplicity of writing turtle and patch behaviours, which usually only requires the subclassing of Turtle and Patch, and the implementation of their `update()` methods (together with any support methods and state variables).

Our framework significantly differs from [6] in its emphasis on distributed programming elements: their framework does not support multiple grids, message passing, patch and turtle mailboxes, turtle migration, or a name server.

The Display class in our framework is considerably more complex than the one in [6]. Message passing is animated: 'envelopes' travel from a sender to a receiver, which may disappear in a 'cloud of smoke' if the communication is unreliable. Turtle migration is displayed as a turtle encased in an 'ice block' which travels between the grids. There is an information output window, which can be written to directly by application code; details are also placed there when the user clicks on entities at run time.

## 4 A Direct Sales Application

In this example, we use the MultiWorldLogo framework to implement a direct sales simulation. It illustrates the use of reliable message passing between different types (subclasses) of turtles, and utilizes turtle mailboxes and migration. In the next section, this application is modified to handle communication in an unreliable setting.

The basic elements of the system are agency members (salespersons) and customers. Agency members obtain their goods from distribution centers. Agency members and customers are allocated to distinct regions.

Agency members hold products which a customer orders from them. If a product runs out, the agency member must visit a distribution center to restock before it can fill a customer's order. A distribution center may be located in a different region from the agency member's, which requires the member to migrate to that region (and back again afterwards).

At start-up time, a customer does not have an assigned agency member. Agency members and customers must travel around their region looking for suitable partners. When an agency member and a customer meet at a patch, the customer will use that agency member as its representative for future purchases.

When a customer orders a product (by message passing), it waits until the product has been sent to it (as a message) by the agency member. After the customer has used a product for a certain period, the product expires, and the customer must order it again.

Agency and Customer are subclasses of Turtle, PatchRegion (the distribution center class) is a subclass of Patch, Region is a subclass of Grid.

Figure 1 shows a typical moment in the execution of the direct sales example. The canvas on the left shows the four grids used in this example (labeled `Grid0-3`, from left to right, top to bottom). Customers are shown as black turtles,

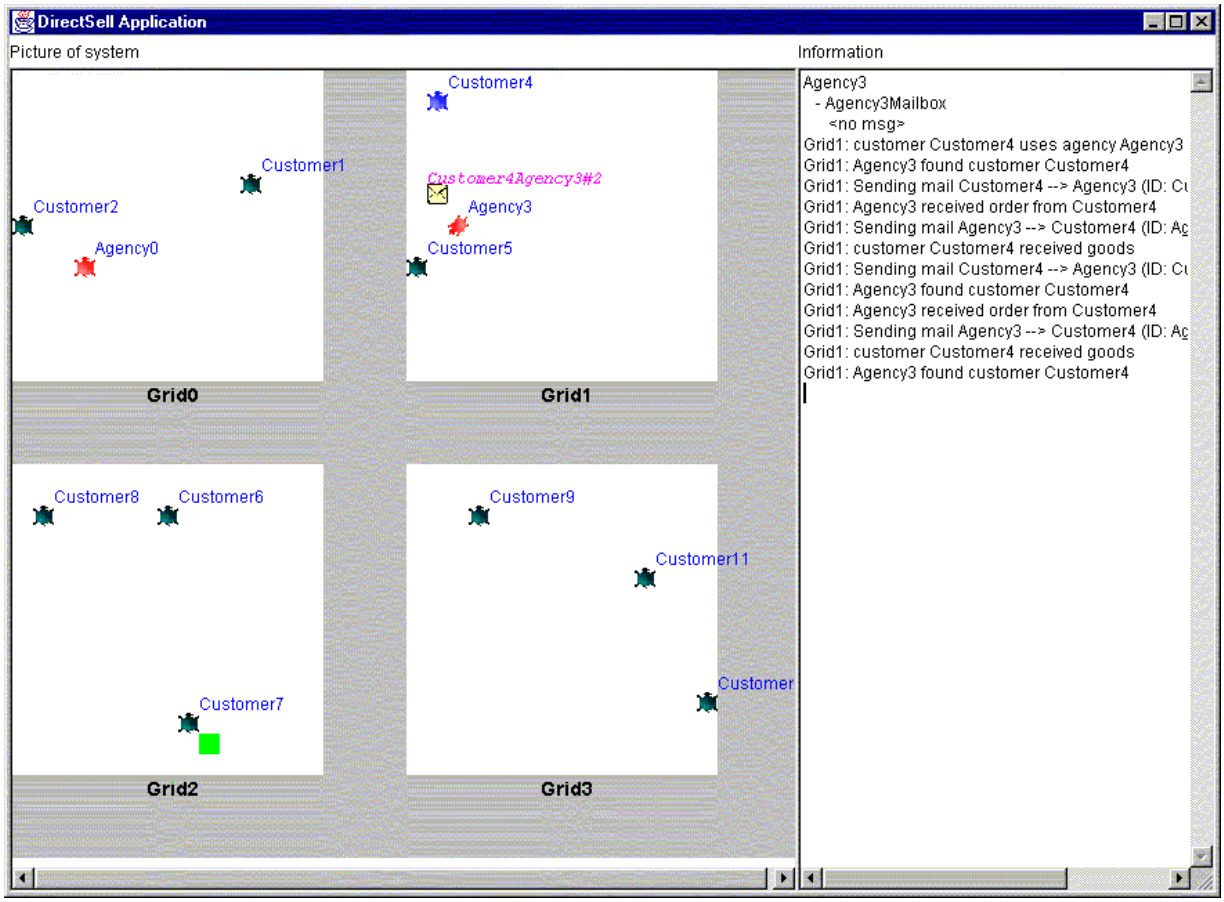


**Figure1. Product Delivery.**

agency members in red (gray). There is only one distribution center (a green (gray) square) in `Grid2`.

The text area on the right details the pattern of communication between the turtles. At the time of the screen shot, `Agency3` in `Grid1` is sending a second product (represented by an 'envelope') to `Customer4`.

We now examine the coding details for the Agency, Customer and PatchRegion classes.

### 4.1. The Agency Class

Agency holds a product amount, and codifies the behavior of an agency member in its `update()` method. If the product amount is greater than zero, the agency member will try to recruit a customer via `findCustomer()`.

Agency will send a product to a customer when it receives an order through `responseMail()`. However, if it has run out of products, it will try to get more by calling `findMoreGoods()`, which causes it to migrate to a region that contains a distribution center.

The Agency code in outline:

```
public class Agency extends Turtle
{       :
  int goodsAmnt = INIT_GOODS_AMNT;
  boolean ready = true;   // has some product
        :
  public void update(){
    if (ready) {          // has some product
      findCustomer();     // find a customer
      responseMail();   // respond to customer
    }
    else          // doesn't have any product
      findMoreGoods();  // goto a dist. center
  }
        :
 } // end of Agency class
```

`responseMail()` extracts an order from the agency's mailbox. It uses `sendGoods()` to send back a product:

```
private void responseMail()
{ PatchTurtle sender;
  // check the mailbox
  while (!(itsMailbox.isEmpty()) && ready) {
    Msg recvMsg = itsMailbox.getMsg();
    String msgType =
          (String)recvMsg.getData();
    sender = recvMsg.getSender();
    if (msgType.equals("order"))
    {    // an order message
      sendGoods( (Customer)sender );
      if (goodsAmnt == 0) {
        ready = false;     // out of product
        setColor(Color.yellow);
      }
  }}
}
```

When the agency member has run out of products, `findMoreGoods()` uses `moveToSellRegion()` to find a distribution center, and migrate to its region:

```
private void moveToSellRegion()
{ Region r;
  int i = 0;
  Vector v = getAllGridRef();
  // find a grid with a dist. center
  do {
    r = (Region)v.elementAt(i);
    i++;
  } while ((!r.hasSellPoint()) &&
           (i < v.size()));
  if (i < v.size())
      migrate(r);   // goto region (grid) r
}
```

### 4.2 The Customer Class

A customer can be in one of three states:

1. It has not yet found an agency member (represented by the constant `STATE_NO_AGENCY`);
2. It has an assigned agency member (represented by the constant `STATE_NORMAL`). This means it can send the agency an order message, but only if the customer currently has no product.
3. The customer has placed an order, and is waiting for a product to be sent (represented by the constant `STATE_WAIT_GOODS`)

After receiving a product, the customer will change back to `STATE_NORMAL`. The 'lifetime' of the product will gradually decrease until it reaches zero, then the customer will place a reorder. The behaviour:

```
public class Customer extends Turtle
{   :
  public void update()
  {
    switch (state) {
      case STATE_NO_AGENCY:
        forward(1);
        break;
      case STATE_NORMAL:         // has agency
        if (timeForGoods == 0) { // no product
          makeOrder();
```

```
          state = STATE_WAIT_GOODS;
          setColor(Color.magenta);
        }
        else         // customer has a product
          timeForGoods--; // decrease 'life'
        break;
      case STATE_WAIT_GOODS:
        String rec;
        if ((rec = receive()) != null) {
          if (rec.equals("goods")) {
          // got a goods message?
            timeForGoods = LIFETIME;
            state = STATE_NORMAL;
            setColor(Color.blue);
          }
        }
        break;
    }
  } // end of update()

  private void makeOrder()
  { // make an order by sending mail
    Msg order =
          new Msg(this,itsagency,"order");
    sendMail(getMailboxOf(itsagency), order);
  }
    :
} // end of Customer class
```

### 4.3. The PatchRegion Class

The PatchRegion class can represent a distribution center. When `hasSellPoint` is true, it checks for the presence of turtles which are agency members and need products:

```
public class PatchRegion extends Patch
{   :
  public void update()
  {
    if (hasSellPoint) { // distribute products
      Vector v = allTurtleHere();
      for (int i = 0; i < v.size(); i++) {
        Turtle t = (Turtle)v.elementAt(i);
        if (t instanceof Agency) {
          // is turtle an agency?
          Agency a = (Agency)t;
          a.setGoodsAmnt(INIT_GOODS_AMNT);
        }
     }}
  }
} // end of PatchRegion class
```

## 5 Direct Sales with Unreliable Communication

This is essentially the same application as described in section 4, except that communication is no longer reliable. The degree of unreliability is controlled by two system-wide variables: one for the rate of message disappearance, the other for the likelihood of reordering.

The application programmer (the student) is now forced to implement different behavior for the agency members and customers to ensure that communication between entities remains reliable.

When a customer places an order with an agency member, it sets a waiting time which decreases with each time step. If it reaches zero and no product has arrived, the customer will reorder the product. When a customer receives a product, it sends an acknowledgment back to the agency member.

The changes to the Agency class are localised in `responseMail()`. An agency member must check whether a message from a customer is a new order or a reorder. This is achieved by using an acknowledgement table to record which customer orders have already been processed. The crucial coding design is to allocate a unique ID to each order message; if an incoming message has the same ID as an earlier one from the same customer then it is a reorder. The agency member responds to a reorder by sending the product again without decreasing its product amount.

## 6 Discussion

MultiWorldLogo is an educational tool to help students grasp the ideas of distributed programming through active learning. The framework consists of Java classes which simulate multiple platforms (grids), message passing and mailboxes, communication in the presence of message loss and reorder, and entity mobility.

Java's object oriented paradigm allows the complexity of distributed programming to be abstracted. Inheritance and polymorphism permit core functionality (e.g. turtles, patches, and grids) to be extended easily. Java's large GUI libraries means that execution has been visualized in several ways. The Java language is simpler than many imperative and object oriented languages, primarily due to its lack of explicit pointers, and its strong error detection.

To date, MultiWorldLogo has not been widely utilized by students since the course in which it would appear ("Client/Server Distributed Systems") is currently offered to students *before*

the Java course. This will be rectified in the next academic year.

The present MultiWorldLogo is a standalone framework which simulates multiple grids. The next version, which is currently under development, is truly distributed. The educational benefit will be that an application becomes the joint work of several students, encouraging group learning. Such an approach highlights problem solving based on cooperation (and conflict) since an application consists of separately designed entities with their own behaviours.

## References


[1] Berson, A. *Client/Server Architecture*, McGraw-Hill, 2nd edition, 1996.

[2] Capretti, G., Cisternino, A., Lagana, M.R., and Ricci, L. "A Concurrent Microworld", In *Proc. of the World Conf. on Educational Multimedia, Hypermedia and Telecommunication, EDMEDIA'99*, 1999.

[3] Capretti, G., Lagana, M.R., and Ricci, L. "Decentralized Programming of Communicating Turtles", In *Proc. of the EUROLOGO'99*, 169-178, 1999

[4] Papert, S. *Mindstorms: Children, Computers, and Powerful Ideas*, Basic Books, 1980.

[5] Resnick, M. *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds*, Bradford Books, The MIT Press, 1995.

[6] Winder, R. and Roberts, G. *Developing Java Software*, John Wiley, 1998.