Preprint (for personal use only); the revised and final version is to appear in the *ITiCSE'02* proceedings.

A Java Toolkit for Teaching Distributed Algorithms

Wolfgang Schreiner
Degree Programme on "Engineering for Computer-based Learning"
Hagenberg University of Applied Sciences
Hauptstraße 117, A-4232 Hagenberg, Austria
+43 7236 3888 2600
Wolfgang.Schreiner@fh-hagenberg.at

ABSTRACT
We present a toolkit for developing and visualizing distributed algorithms in Java. This toolkit consists of a Java class library with a simple programming interface that allows to develop distributed algorithms in a message passing model. The resulting programs may be executed in standalone mode using a Java interpreter or embedded as applets into HTML documents and executed by a Web browser. The toolkit has been applied in various university courses and is freely available.

Categories and Subject Descriptors
K.3.1 [**Computers and Education**]: Computer uses in Education – *Computer-assisted instruction (CAI),* K.3.2 [**Computers and Education**]: Computer and Information Science Education – *Computer science education.*

General Terms
Algorithms, Design, Experimentation.

Keywords
Distributed computing, message passing, Java, applets, visualization, assertions.

1. INTRODUCTION
This paper describes a toolkit for developing and visualizing distributed algorithms in Java [1]. This toolkit was originally named DAJ (Distributed Algorithms in Java), not to be confused with another system with the same name and similar goals which was independently developed at about the same time[2]; to avoid further confusion, we will not refer to this name any more and ultimately chose a different name. The core of our toolkit is a Java application class library with a deliberately simple programming interface which allows to develop distributed algorithms in a message passing model. The resulting programs may be executed in standalone mode or embedded as applets into HTML documents and executed by Web browsers.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ITiCSE'02, June 24-26, 2002, Aarhus, Denmark.
Copyright 2002 ACM 1-58113-499-1/02/0006…$5.00.

Our motivation for this work stemmed from some uneasiness with how to teach distributed algorithms and programming: there are various excellent textbooks on this topic [8,3], but they describe distributed algorithms in an abstract notation rather far away from real programs. Furthermore, there is a lack of an easy to use and universally accessible platform for implementing the algorithms taught in class and investigating their dynamic behavior. Distributed message passing libraries like PVM [5] or systems based on the MPI standard [4] are too "heavy-weight" for use in education; they force to deal with a number of low-level technical details, and they do not allow the easy observation of the "internals" of the execution.

We therefore have designed and implemented a toolkit that provides an easy way of programming distributed algorithms and visualizing their dynamic behavior; the toolkit is based on a programming model that is intuitive but still close to "real" systems. Our toolkit has been implemented using Sun's Java Development Kit. Despite of a few problems related to Java's Abstract Window Toolkit (AWT), the resulting programs should run in any Java environment and can be embedded as applets into HTML pages; this gives the possibility to develop documents that integrate text and executable code in a single framework.

An application developed with the toolkit may run in one of *three modes*:

- standalone, without visualization,
- standalone, with visualization,
- as an applet embedded in a Web page.

The *programming model* has been deliberately kept simple with minimum conceptual overhead:

- The execution model consists of a network of nodes that are partially linked by channels and that operate asynchronously and independently of each other.
- The basic communication mechanism are point-to-point messages between those nodes that are linked by channels.
- Furthermore, a node may send messages via sets of channels and receive (non-deterministically) messages from sets of channels.
- A timeout value may be specified to limit the time that a node is blocked listening on an empty channel (respectively on a set of channels).

- Each node has a local time that is implicitly advanced by each communication operation and that may be accessed by the programmer.

The *visualization* is based on the following elements:

- a screen of user-defined size displays the network as laid out by the user;
- the states of nodes and channels are indicated by colors as well as by messages in a foot line;
- the current local time of each node is displayed;
- the internal states of nodes and channels (i.e., the messages it contains) are exhibited by pop-up windows as defined by the programmer;
- network execution may be slowed down, interrupted, performed step by step, and restarted.

Program execution may be *customized* by

- user-defined node schedulers that determine the next ready node for execution, and by
- user-defined message selectors that determine the next message to be delivered from a channel,

which allows to implement various execution models (synchronous network execution, asynchronous execution) and failure models (lost messages, duplicated messages).

Furthermore, the programmer may state assertions about the *global* state of the network (nodes and channels), which gives the possibility to check the *validity of invariance conditions* which may be formally proved in class.

2. VISUALIZING A PROGRAM

Through the rest of the paper, we will demonstrate the usage of the toolkit by a trivial example; in the last section, we will discuss how the system was received in classroom and list some of the algorithms developed with it. In our demonstration, we are going to implement a distributed program that runs on a ring of three nodes bi-directionally linked to each other. One node emits two messages into both directions; every node iteratively listen for messages from both directions and forwards every received message into the other direction.

The figure on the top of this page illustrates the applet that results from the visualization of this program. The applet window displays three network nodes linked by three pairs of channels; the receiver side of each channel is denoted by a small bullet where the channel touches the node. The nodes contain labels "0", "1", and "2"; the small numerical subscript of each label denotes the current local time of this node; the small number in the left upper corner of the window denotes the global network time.

Pressing the "Run" button starts execution. After the program has terminated, pressing "Reset" initializes the visualization again. "Walk" lets the program run in slow mode, "Interrupt" suspends the program, "Step" allows the program advance one step (i.e., one communication operation). "Redraw" redraws the screen, "Quit" lets the visualization terminate (if the network is currently executing, one has to interrupt it before).

During execution, the current state of each node is represented by the color of its boundary:

Green
the node is ready for execution.

Red
the node is blocked (i.e., it waits for a message on some input channel).

Blue
the node has terminated execution.

Likewise, channel states are denoted by boundary colors:

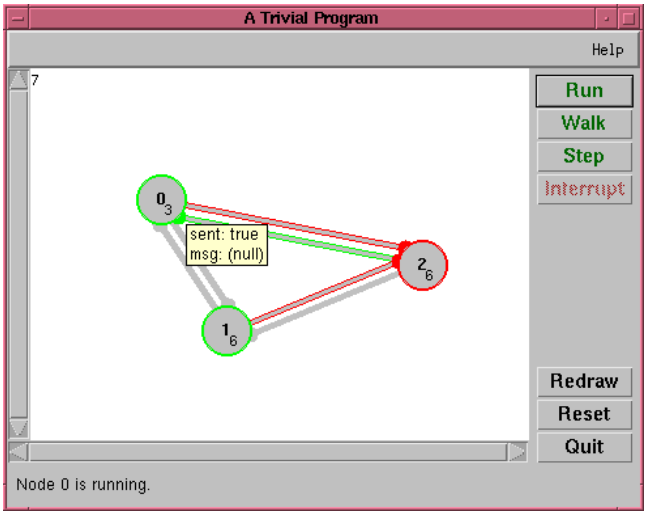
Gray
the channel is empty.

Green
the channel holds at least one message.

Red
the channel is empty and the receiver node waits for a message.

Textual explanations of the different states are given in the bottom line of the window when one moves the mouse cursor over the node respectively channel. Simultaneously, a window pops up that exhibits the internal state of the node respectively channel (in the case of the later, about the queue of messages contained in the channel). A node can be dragged to a different location by clicking on it and moving the mouse while keeping the mouse button pressed.

Selecting in menu "Help" one of the items "About Algorithm" or "About Toolkit" lets some informative panels pop up. Selecting "Home", "Copying", or "Help" loads the corresponding Web page into a window of the Web browser.



3. DEVELOPING THE PROGRAM

The program described in the previous section has been implemented by the following pieces of source code compiled with the classes of the toolkit.

```
public class Main extends Application
{
  public static void main(String[] args)
  {
    new Main().run();
  }

  public Main()
  {
    super("A Trivial Program",
      400, 300);
  }

  public void construct()
  {
    Node node0 = node(new Prog(0),
      "0", 100, 100);
    Node node1 = node(new Prog(1),
      "1", 150, 200);
    Node node2 = node(new Prog(2),
      "2", 300, 150);
    link(node0, node1);
    link(node1, node2);
    ...
  }
}
```

An application is represented by an object of a subclass of class `Application`. The programmer provides in this class a method `construct` that creates the nodes of the network and defines the corresponding channel connections. Each network node is created by a call of the function `node` that takes as its arguments the program executed by the node, the node label shown by the visualizer and the coordinates of the node in the visualization (measured in pixels from the left upper corner of the visualization area). The default constructor of the class determines the appearance of the visualizer by the title displayed in its frame and by the size of the visualization area. If the program shall be also executable in standalone mode, the class must also contain a method `main()` that creates the application object and starts its execution by invoking the method `run`.

Having defined the network, we now are going to implement the program executed by each network node:

```
class Prog extends Program
{
  int number;

  public Prog(int i)
  {
    number = i;
  }

  public String getText()
  {
    String msgString;
    if (msg == null)
      msgString = "(null)";
    else
      msgString = msg.getText();
    return
      "sent: " +
      String.valueOf(sent) +
      "\nmsg: " + msgString;
  }

  public void main()
  {
    if (number == 0)
      {
        out(0).send(new Msg(0));
        out(1).send(new Msg(1));
      }
    for (int i = 0; i < 5; i++)
      {
        int index = in().select();
        Message msg =
          in(index).receive();
        out(index).send(msg);
      }
  }
}
```

A node program is an object inheriting from the toolkit class `Program`. In above example, we define a constructor that initializes the program with its index (in order to let the program exhibit different behaviors on different nodes). The method `main` represents the code actually executed by the network node. In our example, node 0 sends two messages to its output channels with predefined names `out(0)` and `out(1)`. Each node runs through three iterations in which it

1. determines which of its input channels is not empty
   `index = in().select()`
2. receives the message contained in this channel
   `msg = in(index).receive()`
3. and sends it to the opposite output channel.
   `out(index).send(msg)`

A node type should override the method `getText` to exhibit the internal state of each node; the visualizer calls this method when displaying the node state in a pop-up window.

What now remains to be defined is the type of messages sent respectively received via the channels of the network:

```
class Msg extends Message
{
  int val;
  public Msg(int i)
  {
    val = i;
  }
  public int value()
  {
    return val;
  }
  public String getText()
  {
    return "content = " +
      String.valueOf(val);
  }
}
```

A message type extends the class `Message` and should provide a method `getText` that returns the content of the message in a single line. This method is invoked by the visualizer when it displays the contents of a channel in a pop-up window.

4. MAKING THE APPLET

An application developed with the toolkit can be embedded as an applet into a Web page using the `APPLET` tag. The applet can be customized by the parameters specified below:

```
<APPLET code="C.class" width=w height=h>
  <param name=buttonLabel value=string>
  <param name=fontName value=string>
  <param name=fontStyle value=string>
  <param name=fontSize value=int>
</APPLET>
```

The applet loads class *C*`.class`; it appears as a button of width *w* and height *h*. The default label on the button is the title specified in the constructor of the corresponding `Application` object; this label may be overridden by the applet parameter `buttonLabel`. The font used for the button label may be customized using the optional applet parameters `fontName`, `fontStyle`, and `fontSize` which determine the corresponding Java font.

5. ASSERTING CONDITIONS

While only the code explained above is required to let the program run, we also would like to make explicit that the program fulfils certain properties. The most important technique for reasoning about distributed algorithms is to find *invariance conditions* that describe the state of the system in every step in every possible execution [9]. Our toolkit allows to formulate such a condition as an *assertion* that is checked by the simulator in the actually performed execution.

In above example, a central property is that there are exactly two messages contained in the network. We can state this as an assertion by rewriting class `Prog` as follows:

```
class Prog extends Program
{
  private int number;
  public int index;
  public Message msg = null;
  public boolean sent = false;

  public Prog(int i)
  {
    number = i;
  }

  public void main()
  {
    if (number == 0)
      {
        out(0).send(new Msg(0));
        out(1).send(new Msg(1));
        sent = true;
      }

    GlobalAssertion assertion =
      new NumberOfMessages();

    for (int i = 0; i < 5; i++)
      {
        assert(assertion);
        index = in().select();
        msg = in(index).receive();
        out(index).send(msg);
        msg = null;
      }
  }
}
```

This code differs from the previous one by making `index` and `msg` public instance variables of class `Prog` and by introducing an boolean instance variable `sent` which is set by the program on node 0 to true when it has initialized the network. Furthermore, we reset the received message `msg` to null after it has been submitted again into the network.

When `assert` is called with an assertion *a* as its argument, it invokes the method *a*`.assert(`*p*`)` on the array of node programs *p*. The assertion may examine the published state of each program object (in our example, the contents of the public instance variables `index`, `message`, and `sent`). It may also investigate the contents of the attached channels (`in`, `out`) by calling the method `getMessages` which returns the array of messages contained in a channel (in the order in which they were sent). If the method returns false, execution is aborted with the error message returned by *a*`.getText()` (which is displayed in the foot line of the visualization window).

The assertion type `NumberOfMessages` is defined as a subtype of the type `GlobalAssertion` as shown below:

```
class NumberOfMessages extends
  GlobalAssertion
{
  int count;

  public String getText()
  {
    return "invalid no of msgs: " +
      String.valueOf(count);
  }

  public boolean assert(Program prog[])
  {
    if (!((Prog)prog[0]).sent)
      return true;
    count = 0;
    for (int j = 0; j<prog.length; j++)
      {
        Prog program = (Prog)prog[j];
        count += getMessages(
          program.out(0)).length;
        count += getMessages(
          program.out(1)).length;
        if (program.msg != null)count++;
      }
    return count == 2;
  }
}
```

The assertion says that, as soon as node 0 has signaled by its variable `sent` the initialization of the network, there must be exactly two messages in the network. A message may be either contained in some channel `out(`*j*`)`, or, in a program's local variable `msg`.

6. CLASSROOM APPLICATION

Since 1997 we have applied the toolkit in four graduate classes on distributed algorithms. After a short introduction to the use of the toolkit by the example demonstrated above, the students (most of which had some previous experience with Java) could use the toolkit without much further help.

We have mainly used the toolkit in two modes of operation:

1. Exercise work: we present in class an algorithm in a formal model such as I/O automata [8] and let the students implement the algorithm with the help of the toolkit as a standalone application.
2. Project work: at the end of the course, we hand out textbook material and let the students (in groups of 2 or 3 members) study algorithms on their own in order to prepare a classroom presentation for their colleagues. The students set up a Web page which explains the algorithm, cites the relevant literature, embeds the applet and contains a link to the source code of the implementation.

In our experience, the integration of theoretical studies with exercises on the development of executable code has been very successful. In particular, it has increased the motivation of students to study algorithmic details which are necessary for an actual implementation.

Mainly from the results of the project work, we have gathered a small library of approximately 15 algorithms implemented with the toolkit (see the URL in [1]), e.g.

- Distributed Snapshots
- Parallel Convex Hull Construction
- Termination Detection
- Breadth First Search
- Invitation Algorithm
- Maekawa's Mutual Exclusion Algorithm
- LyHudak Mutual Exclusion
- RicartAgrawala Mutual Exclusion
- Dijkstra Scholten Termination Detection
- Total Order Broadcasting

This collection can be used in subsequent courses to demonstrate various aspects of the algorithms by classroom demonstrations.

7. RELATED WORK

The work closest in spirit to our own developments is the DAJ software described in [2]. Here a distributed system is simulated by a collection of Java applets that are embedded in the same HTML document. Each applet implements by a state machine one node of the system; initially the applets use the standard Java API to learn about each other (`getAppletContext()`). After that, an applet *A* may send a message to some applet *B* by invoking a method `receive()` in *B*. While the user-interface is application-dependent, the algorithm-independent part of the simulation is provided by the DAJ class DistAlg. The system has been used for the implementation of several fundamental distributed algorithms.

While this software and our toolkit share goals (and accidentally also the name), we have chosen a different implementation strategy: a distributed system is simulated by a multi-threaded application that relies on a central scheduler. Each system node is represented by a separate thread which returns control to the scheduler each time that a message passing operation is performed. The program executed by each node is a conventional program written in an imperative message passing style (not as a state machine) such that the programming experience is close to that of writing a PVM or MPI program. Furthermore, our toolkit provides a graphical user interface which allows to investigate the states of nodes and channels by point-and-click operation.

A more recent development is the Distributed Algorithm Platform (DAP) which is currently being developed on the basis of the LEDA library [10]. DAP will have a GUI through which the user will be able to control the simulation of his/her algorithm. The users should be able to use the GUI to define the network topology, enter initial condition, generate events for the simulation, etc.

8. ACKNOWLEDGEMENT

I thank Prof. Mordechai Ben-Ari who became aware of my toolkit and encouraged its publication. The toolkit was developed at the Research Institute for Symbolic Computation (RISC-Linz) of the Johannes Kepler University in Linz, Austria.

9. REFERENCES

[1] Wolfgang Schreiner. DAJ -- A Toolkit for the Simulation of Distributed Algorithms in Java. Technical Report 97-36, Research Institute for Symbolic Computation (RISC-Linz), Johannes Kepler University, Linz, Austria, November 1997. http://www.risc.uni-linz.ac.at/software/daj.

[2] Mordechai Ben-Ari. *Distributed Algorithms in Java*. 2nd SIGCSE/SIGCUE Conference on Integrating Technology into Computer Science Education, Uppsala, Sweden, 1997, pp. 62-64.

[3] Chandy Chow and Theodore Johnson. *Distributed Operating Systems & Algorithms*. Addison-Wesley, Reading, Massachusetts, 1997.

[4] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard (Version 1.2), 1995. http://www.mpi-forum.org/docs/docs.html.

[5] Al Geist et al. PVM: Parallel Virtual Machine -- A Users' Guide and Tutorial for Networked Parallel Computing, 1994. http://www.epm.ornl.gov/pvm/pvm_home.html.

[6] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Sun Microsystems, 1.0 ed., 1996. http://www.javasoft.com/docs/books/jls/html/.

[7] JavaSoft. Java Development Kit -- Version 1.1.4, 1997. http://www.javasoft.com/products/jdk/1.1/.

[8] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, CA, 1996.

[9] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems : Safety*. Springer, Berlin, Germany, 1995.

[10] LEDA Extension Package: Distributed Algorithms Platform (DAP). http://faethon.cti.gr/lep-dap.