DAJ — A Toolkit for the Simulation of Distributed Algorithms in Java*

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, A-4040 Linz, Austria
Wolfgang.Schreiner@risc.uni-linz.ac.at
http://www.risc.uni-linz.ac.at/people/schreine

November 14, 1997

Abstract

This reports describes the DAJ toolkit for designing, implementing, testing, simulating, and visualizing distributed algorithms in Java. DAJ 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 pages and executed by Web browsers. The goal of the toolkit is to provide an universally accessible platform for research and education in the area of distributed algorithms. DAJ is freely available over the World Wide Web.

*The HTML version of this document and the described software are available at http://www.risc.uni-linz.ac.at/software/daj. We recommend to read the much better structured HTML version rather than the paper version.

Contents

1 Introduction

This reports describes the DAJ toolkit for designing, implementing, testing, simulating, and visualizing distributed algorithms in Java [4]. DAJ consists of a Java class library with a deliberately simple programming interface that allows to develop distributed algorithms in a message passing model. The resulting programs may be executed in standalone mode by a Java interpreter or embedded as applets into HTML pages and executed by available Web browsers.

Our motivation for this work stems from some uneasiness with how to teach distributed algorithms and programming: there are various excellent textbooks on this topic [6, 1], 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 dynamical behavior. Distributed message passing libraries like PVM [3] or systems based on the MPI standard [2] 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 implement the DAJ toolkit that provides an easy way of programming distributed algorithms and visualizing their dynamic behavior based on a programming model that is intuitive but still close to "real" systems. DAJ has been implemented using Sun's Java Development Kit [5]. Despite of a few problems related to Java's Abstract Window Toolkit (AWT), the resulting programs should run in any Java environment and be embeddable 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 DAJ 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 channel sets and receive (non-deterministically) messages from channel sets.
- A timeout value may be specified to limit the time that a node is blocked listening on an empty channel (or channel set).

- 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.

The structure of this report is as follows:

Section 2 is a tutorial on the use of the toolkit by a simple example.

Section 3 describes the installation of DAJ and related issues.

Section 4 is a reference manual on the construction of applications in DAJ by composing nodes to networks.

Section 5 is a reference manual on the writing of node programs in DAJ running on individual network nodes.

Appendix A sketches how to proof assertions about programs designed and implemented in DAJ.

Appendix B contains an example of a non-trivial distributed algorithm designed and implemented in DAJ.

The DAJ toolkit is freely available at the URL `http://www.risc.uni-linz.ac.at/software/daj`

Reports, comments, and suggestions should be addressed to the author at

`Wolfgang.Schreiner@risc.uni-linz.ac.at`

2 Using the Toolkit

This section explains the usage of the toolkit by a trivial example: we are going to implement the simulation of a distributed program that runs on a ring of three nodes bidrectionally 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.

2.1 Executing the Program

If you view this document in HTML format on our Web page, you should see a button labelled "Press Me" embedded below. If you do not see this button but the string "Please enable Java!" instead, you must enable Java on your Web browser[1] and then reload the page by pressing the "Reload" button.

In case your browser does not support Java or you read this document in text form, we include an image of the window in Figure 1.

Do as the "Press Me" button says; you should see a window pop up that visualizes a program implementing the example described above. The 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, press "Reset" to initialize 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, you have 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 (waiting for a message on some channel).

Blue the node has terminated execution.

[1] In Netscape Communicator 4.0, select the menu item `Edit/Preferences/Advanced/-Enable Java`.

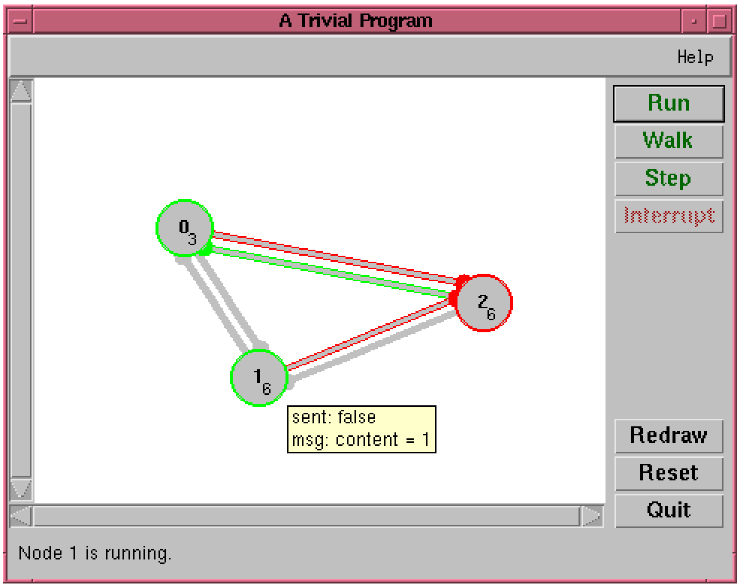

Figure 1: A Trivial Program

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 you move 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 your Web browser.

2.2 Writing 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 DAJ toolkit.

```
import daj.*;

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);
    link(node2, node0);
    link(node0, node2);
    link(node2, node1);
    link(node1, node0);
  }

  public String getText()
  {
    return "A Trivial Program\n \n" +
      "2 messages circling bidirectionally\n" +
      "through a ring of three nodes";
  }
}
```

The DAJ toolkit is contained in a Java package `daj`; this package has to be imported in the beginning. An application is represented by an object of a subclass of the DAJ class `Application`. The programmer has to provide 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 (non-applet) mode, the class must also contain a method `main()` that creates the application object and starts its execution by invoking the method `run`.

The method `getText` returns the text (a sequence of lines separated by newlines) that it displayed in a window when the user selects the item "About Algorithm" in menu "Help".

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 DAJ class `Program`. In above example, we define a constructor that initializes the program with its index (in order to let the 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 must inherit from DAJ 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.

2.3 Asserting Conditions

While only the code explained in the previous section is required to make the program executable, we would like to make explicit that the program fulfils certain properties. The central property of above example 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;
  public boolean sent;

  public Prog(int i)
  {
    number = i;
    msg = null;
    sent = false;
  }
```

```
  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 DAJ type **GlobalAssertion** as shown below:

```
class NumberOfMessages extends GlobalAssertion
{
  int count;

  public String getText()
  {
    return "invalid number of messages: " +
      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 signalled 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`.

When `assert` is called, it may rely on the fact that each program passed to it as its argument is either in the initial state, has terminated execution, or is blocked on a communication operation *before* the corresponding message was sent or received[2]. This allows to formulate assertions about the state of the network in a comfortable way.

Appendix A contains a formal proof of this assertion.

2.4 Compiling the Program

Assuming that all of the code presented in the previous section is contained in a file *Main.java*, that the DAJ toolkit is appropriately installed (see Subsection 3), and of course that Java is running on your system, you can compile the program by executing

```
javac Main.java
```

The program may be then called in stand-alone mode by calling

```
java Main
```

By creating a HTML file *Main.html* and inserting the code

```
<APPLET code="Main.class"
  width=120 height=30
  alt = "Your browser does not support Java.">
  <param name=buttonLabel value="Press me">
Please enable Java!
</APPLET>
```

you can embed the application as an applet in a Web page (see Subsection 2.5). Calling e.g.

```
appletviewer Main.html
```

allows you to test the applet.

[2]There is one additional case: the program may have explicitly interrupted execution by a call of `yield` (see Subsection 5.5)

2.5 Writing Applets

An application developed with the DAJ 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>
Please enable Java!
</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 (see Subsection 4.1); this label may be overridden by the applet parameter `buttonLabel`.

The font used for the button label may be customized using the applet parameters `fontName`, `fontStyle`, and `fontSize`, each of which is optional and must determine a corresponding Java font.

When writing applet code, please keep in mind that most current commercial browsers (e.g., Netscape Communicator 4.0) only provide reliable support for Java 1.0.x. *Do not yet use Java 1.1.x if you want to write applets that can be executed by a variety of browsers.*

For installing an HTML file including a DAJ applet on a Web server, one has to make sure that the corresponding DAJ classes are appropriately installed and served:

1. Java 1.0.x: install directory *daj* (with subdirectory *awt*) in some location of the Web server and make sure that all *.class* files in these directories are served with Content-Type `application/octet-stream`.

   E.g., under the WN server, this is achieved by putting a file *index* into directories *daj and daj/awt* with contents

   ```
   Default-Content=application/octet-stream
   Attribute=serveall
   ```

   and calling `wndex` within *each* of both directories.

   The applet has then to specify the attribute `codebase` to denote the URL of directory *daj*, i.e.,

   ```
   <APPLET code="C.class"
     codebase="url"
     width=w height=h>
   </APPLET>
   ```

   You may omit the `codebase` attribute only if you install the directory *daj* in the directory of the HTML file containing the applet.

2. Java 1.1.x: as explained above, Java 1.1 is not yet properly supported by most current commercial browsers.

   However, when this support becomes available, the programmer may combine all the newly defined Java classes of his DAJ application into a Java archive (JAR) file *a.jar* by calling e.g.

   ```
   jar cvf a.jar *.class
   ```

   This archive has to be installed together with the DAJ archives *daj.jar* and *awt.jar* in a directory served by the Web server making sure that these files are served with Content-Type `application/java-archive`.

   E.g. under the WN server, this is achieved by putting a file *index* into this directory with contents

   ```
   Attribute=serveall
   Default-content=application/java-archive
   ```

   and calling `wndex` within this directory.

   The applet then has to specify the attributes `codebase` and `archive` to denote the URL of the directory containing the JAR files and the names of the files, respectively:

   ```
   <APPLET code="C.class"
     codebase="url" archive="a.jar, daj.jar, awt.jar"
     width=w height=h>
   </APPLET>
   ```

3 Installing the Toolkit

3.1 Copying Conditions

DAJ was developed by the author of this document who is the *only* owner of *all* parts of the toolkit.

The toolkit is in its binary form available for free use provided that the use of the toolkit is properly cited in resulting applications and publications. Further distribution is permitted under the constraint that the *whole* set of class files is distributed *without* modification.

I'd appreciate if you would send me any programs that you have written using the toolkit (binary and, if possible, source code) plus any accompanying documentation or publications.

3.2 Source Code

The source code of the DAJ toolkit is generally *not* available for free inspection or modification. If you encounter any problems or bugs, please notify me. If you have suggestions for extensions in functionality, please tell me. I probably do not have time to develop the extensions myself, but I may give you access to the source code to incorporate the functionality under conditions to be negotiated.

If the toolkit does not work properly in some Java environment and you would like to make it run there, please send me an email describing the problem. I may then give you access to the source code under conditions stated on the toolkit's Web page.

3.3 Binary Code

The DAJ toolkit comes in two versions, one for JDK 1.0.x, one for JDK 1.1.x. Calling

```
java -version
```

on your installation machine tells you which version you need. However, if you want to serve your program also as an applet (see Subsection 2.5), you should use the DAJ version for JDK 1.0.x *and* also use JDK 1.0.x for compilation (many popular browsers do not support JDK 1.1).

Installing the DAJ toolkit is simple. Download the files *daj.zip* and *awt.zip* for the appropriate JDK version and set the environment variable `CLASSPATH` to include the paths of these files. This allows you to use DAJ locally.

If you want to serve DAJ applets compiled with the 1.0 version, you have to use `unzip` to extract the individual *.class* files from *daj.zip* and install them in some directory *daj*[3]. Likewise, the *.class* files from *awt.zip* are extracted and installed in directory *daj/awt*. This directory has to be served by your Web server as explained in the previous section.

If you want to serve DAJ applets compiled with the 1.1 version (remember that they will not run in many browsers), you may proceed as with the 1.0 version or, alternatively, download and install the files *daj.jar* and *awt.jar* in some directory served by your Web server as explained in the previous section.

3.4 Known Problems

The toolkit was developed and tested under Linux, Solaris, Windows NT 4.0 using JDK 1.0.2 (Linux, Solaris), JDK 1.0.3 (Linux), JDK 1.0.4 (Solaris), Netscape Navigator 3.02 (Linux, Solaris), Netscape Communicator 4.03 (Linux, Solaris, Windows NT 4.0). I know about the following user interface problems but do not know how to fix them because of bugs or limitations (features?) of the different versions of the Java AWT.

JDK PopUp Windows in Applets All versions of JDK aparently have serious bugs in class `Window` in applet mode, which makes it impossible to determine the appropriate size of a popup window and to display the

[3] If you are running Unix and do not have access to `unzip`, extract the class files from the GNU zipped tar archives that can be also found on the DAJ Web page.

contents such that they are visible under different browsers. The applet warning string is displayed at different edges of the window (top, bottom, *left*) by different browsers *without* adjusting the window size or coordinate system.

DAJ for JDK 1.0 uses a hack that should make the contents visible under most browsers (JDK 1.0 appletviewer, Netscape 3.0/4.0 under Linux, Solaris, and Windows NT 4.0) but yields oversized windows. DAJ for JDK 1.1 uses a hack that makes the contents visible under JDK 1.1 appletviewer and gives even more ugly results.

JDK 1.1 Scrolling In JDK 1.1, positioning scrollbars apparently behaves differently from JDK 1.0, therefore the DAJ source is different in both versions. Scrolling the visualization area in an application developed with JDK 1.0 does not work correctly under JDK 1.1 (and certainly not vice versa).

JDK 1.0 X-Server Memory Consumption In JDK 1.0 under X-Windows, during a visualization run, the memory consumption of the X-Server continuously grows until the visualization is interrupted (by which free memory becomes instantly available again). This may bring especially X-Terminals down to their knees during longer visualization runs.

JDK 1.0 Scrolling in Applets In JDK 1.0, on various browsers the size of the vertical scrollbar slider does not match the size of the visualization area. In Netscape 4.0 on Solaris, the slider is one pixel too small, in Netscape 4.0 on Windows NT 4.0, the slider is about a frame width too small.

JDK 1.0 PopUp Window Positions In JDK 1.0 the computation of a component's offsets from its parent component apparently does not work correctly. The pop-up windows do not appear in the specified locations but a few pixels off.

JDK 1.0 Window Resizing In JDK 1.0, resizing the visualization window returns a zero coordinate in the corresponding event. Thus there is apparently no possibility to determine the correct window position, if the window is resized from the left upper corner. Popup windows appear in wrong places until the window is moved.

JDK 1.0 Window Closing In JDK 1.0, it is apparently not possible to trigger an action when the user selects the menu item "Close" in a window's frame. The window remains open.

JDK 1.0 Visualization Window in Applets In JDK 1.0, there is apparently no possibility to determine the position of an applet in absolute screen coordinates. The visualization window thus pops up always in a fixed place.

JDK 1.0 Help Menu In JDK 1.0 on Linux, setting help menus for menu bars does not work. Thus the help menu is generally added as a normal left-justified menu in DAJ for JDK 1.0.

JDK Scrollbars In both versions of JDK, it is apparently not possible to determine the height/width of a scrollbar before the component is created to which the scrollbar is attached. Thus the visualization area is a bit smaller than specified by the programmer in both dimensions (the area includes both the horizontal and the vertical scrollbar).

Netscape 4.0 Applet Redraw Under Netscape 4.0 on Windows, the visualization area of an DAJ applet is sometimes not redrawn when the popup window is closed.

Netscape 4.0 Applet Stops Netscape 4.0 does not trigger the `stop` method of an applet when the browser is iconified, thus the visualization window remains open.

3.5 Release Information

This document describes version 1.0.1 of the toolkit.

DAJ 1.0.1 Add `Scheduler.incTime()` and show the global network time in the visualization window.

DAJ 1.0 Base version.

4 Building Applications

An application is constructed by providing an object of type `Application` (see Subsection 4.1). The definition involves

- constructing the network (see Subsection 4.2),
- (optionally) customizing the network execution (see Subsection 4.3), and
- (optionally) customizing the network visualization (see Subsection 4.4).

4.1 Types

`Application`: Network Applications

```
abstract class Application
{
  // constructing the application
  Application()
  Application(String t, int w, int h)

  // called for constructing the network
  abstract void construct()

  // used for standalone mode
```

```
  void run()

  // used for applet mode
  String getText()

  // to be used by construct()
  Node node(Program p)
  Node node(Program p, String t, int x, int y)
  void link(Node n1, Node n2)
  void link(Node n1, Node n2, Selector s)

  // customizing the execution
  void setScheduler(Scheduler s)
  void setSelector(Selector s)

  // customizing the visualization
  void setNodeRadius(int r)
  void setNodeFonts(Font f1, Font f2)
  void setChannelWidth(int w)
  void setBrowser(String b)
}
```

An object of (a non-abstract subtype of) this type is the starting point of program execution and visualization. The default constructor creates an application without visualization (usable only in standalone mode), the other constructor creates an application visualized by a window titled t and a visualization area of width w and height h[4].

The programmer must override the method `construct` to create and to link the nodes of the corresponding network.

If the application shall be usable in standalone mode, the programmer must create an object of this type and call its method `run`, e.g.,

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

If the application shall be usable as an applet, the programmer must override its default constructor to call the constructor with title, width, and height:

[4]Due to a limitation of the Java AWT, the useable size of the visualization area is a bit smaller since this area *includes* both the horizontal and the vertical scrollbar

```
public class Main extends Application
{
  ...
  public Main()
  {
    super("Title", 400, 300);
  }
}
```

The method `getText` returns an applet information string that can be displayed by the applet viewer or Web browser (default "`(no information)`").

The other methods may be used for constructing the network, customizing the execution, and customizing the visualization (see the following subsections)¿

`Node`: Network Nodes

```
class Node
{
}
```

A node is an element of a network; it may be created (only) by a call of function `node`. Nodes may be linked via communication channels by the function `link`.

`Scheduler`: Program Schedulers

```
abstract class Scheduler
{
  int getNumber()
  boolean isReady(int i)
  int incTime();
  abstract int nextProgram()
}
```

A scheduler selects the next program for execution among those node programs that are in the *ready* state. A scheduler must redefine the abstract function `nextProgram()` such that this function returns some i with $0 \leq i <$ `getNumber()` and `isReady(i)` (i.e. it must select one ready node program for execution). If this condition cannot be fulfilled, the scheduler must return -1 (denoting deadlock). The scheduler should call `incTime()` on each attempt to select a (possibly non-ready) program for execution; this will increase the global network time in the upper left corner of the visualization window.

The default scheduler selects in a round robin mechanism the next program ready for execution; it is an object of type

```
class SchedulerDefault extends Scheduler
{
```

```
    int last = -1;
    int nextProgram()
    {
      int n = getNumber();
      boolean reset = false;
      do
        {
          incTime();
          last++;
          if (last == n)
            {
              last = 0;
              if (reset) return -1;
              reset = true;
            }
        }
      while (!isReady(last));
      return last;
    }
  }
```

The default scheduler may be redefined (see Subsection 4.3) to implement different kinds of network models (synchronized networks, asynchronous networks, ...).

`Selector`: Message Selectors

```
  interface Selector
  {
    Message select(MessageQueue queue);
  }
```

A selector returns the next message to be delivered from a channel from the queue of messages (see Subsection 4.1) stored in the channel; it also updates the queue.

The default selector returns the oldest message in a channel (i.e., the message which was sent first); it is an object of type

```
  class SelectorDefault implements Selector
  {
    Message select(MessageQueue queue)
    {
      return queue.dequeue();
    }
  }
```

The default selector may be redefined (see Subsection 4.3) to implement different kind of channel failure modes (lost messages, duplicated messages, ...).

`MessageQueue`: Message Queues

```
class MessageQueue
{
  MessageQueue()
  boolean isEmpty()
  void enqueue(Message m)
  Message dequeue()
  MessageCell getCell()
}
```

This type is used to implement message selectors (see Subsection 4.1).

A message queue is a FIFO (first in/first out) list of messages. The default constructor creates an empty queue; `isEmpty` returns true if and only if a queue is empty; `enqueue` adds a message to the end of the queue; `dequeue` extracts the head of the queue (removing it from the queue).

For low-level queue manipulation, `getCell` returns the cell (see Subsection 4.1) representing the head of the queue (or `null`, if the queue is empty).

`MessageCell`: Message Cells

```
class MessageCell
{
  MessageCell(Message m)
  MessageCell(Message m, MessageCell c)
  Message getMessage()
  MessageCell getNext()
  void setMessage(Message m)
  void setNext(MessageCell c)
}
```

This type is used for low-level access to message queues (see Subsection 4.1).

A message cell contains a message and a “next” cell (which may be `null`). The unary constructor constructs a cell with message m and sets the next cell to `null`; the binary constructor constructs a cell from given m and c. `getMessage` returns the message from the cell; `getNext` returns the next cell. `setMessage` sets the cell's message to m; `setNext` sets its next cell to c.

4.2 Constructing the Network

The following functions may be used (*only*) by the method `construct` of an `Application` to define the network where the application is executed.

`node`: Creating Nodes

```
n = node(p)
  Node n
  Program p
n = node(p, t, x, y)
  Node n
  Program p
  String t
  int x
  int y
```

Both functions create a new network node n executing the program denoted by p.

The first function may be used only if the program is *not* visualized (see Section 4.4).

The second function tells the visualizer to label the visual representation of n with text t and put it at the screen coordinates x/y. This function may be also used if the program is not visualized.

`link`: Linking Nodes

```
link(n1, n2)
  Node n1
  Node n2
link(n1, n2, s)
  Node n1
  Node n2
  Selector s
```

The first function creates a uni-directional point-to-point channel from $n1$ to $n2$ whose messages are delivered using the default selector (see Subsection 4.1). The set of output channels `out` of $n1$ is extended by one element representing the sender side of the channel; likewise the set of input channels `in` of $n2$ is extended by one element representing the corresponding receiver side.

The second function behaves like the first one except for the fact that selector s is used for the delivery of messages from this channel.

4.3 Customizing the Execution

The following functions may be used (*only*) by the constructor method of an `Application` to customize the behavior of network execution.

`setScheduler`: Setting the Program Scheduler

```
setScheduler(s)
  Scheduler s
```

This function redefines the default program scheduler (see Section 4.1) as *s*.

`setSelector`: Setting the Message Selector

```
setSelector(s)
  Selector s
```

This function redefines the default message selector (see Section 4.1) as *s*.

4.4 Customizing the Visualization

Visualization takes part in a screen area defined by the `Application` object (see Subsection 4.1). Various aspects of the visualization may be customized by the following functions to be called *only in the constructor of the application object.*

`setNodeRadius`: Set the Node Radius

```
setNodeRadius(r)
  int r
```

Each node is displayed as a filled circle. This function may be used to redefine the radius of this circle as *r*.

`setNodeFonts`: Set the Node Fonts

```
setNodeFonts(f1, f2)
  Font f1
  Font f2
```

The node label is displayed with a subscript denoting the local time of the node. This function may be used to redefined the font of the label as *f1* and the font of the subscript as *f2*.

`setChannelWidth`: Set the Channel Width

```
setChannelWidth(w)
  int w
```

Each channel is represented by a line. This function may be used to redefine the width of this line as *w*.

`setBrowser`: Set the Web Browser

```
setBrowser(b)
  int b
```

In standalone mode, a Web browser is invoked when the user selects the menu item "Home", "Copying", or "Help" to display the content of the corresponding Web page. This function may be used to redefine the browser command (default "`/usr/local/bin/netscape`") as *b*.

5 Programming Nodes

For each node of the network (see Subsection 4.2) the corresponding program has to be defined. This program may

- access the node interface (see Subsection 5.2),
- communicate with other nodes (see Subsection 5.3),
- communicate with groups of other nodes (see Subsection 5.4), and
- control its execution in various other ways (see Subsection 5.5).

5.1 Types

`Program`: Node Programs

```
abstract class Program
{
  // to be overridden by the user
  abstract void main()
  String getText()

  // the node's communication interface
  InChannelSet in()
  InChannel in(int i)
  OutChannelSet out()
  OutChannel out(int i)

  // program execution control
  void assert(GlobalAssertion a)
  int getTime()
  void sleep(n)
  void yield()
  void exit(int i)
}
```

Each network node holds an object of some (non-abstract) subtype of `Program`. The method `main` must be overridden by the programmer; it represents the program being executed by that node.

The method `getText` returns a string (default: "`(no information)`") that consists of a sequence of lines separated by the newline character. This text is displayed by the visualizer in a pop-up window whenever the mouse pointer is moved over the visual representation of the node. The programmer may override this method to exhibit the internal state of the node.

The other methods may be used by `main` for communication with other nodes and for controlling the execution of the node, respectively. Their behavior is documented in the following subsection.

`InChannel`: Input Channels

```
class InChannel
{
  Message receive()
}
```

An object of this type represents a channel from which a message may be received.

`OutChannel`: Output Channels

```
class OutChannel
{
  void send(Message m)
}
```

An object of this type represents a channel to which a message may be sent.

`InChannelSet`: Sets of Input Channels

```
class InChannelSet
{
  InChannnelSet()
  int getSize()
  void addChannel(InChannel c)
  InChannel getChannel(int i)
  int select()
  int select(int t)
}
```

An object of this type represents a set of input channels. The default constructor creates an empty set. The other methods are described in the following subsection.

`OutChannelSet`: Sets of Output Channels

```
class OutChannelSet
{
  OutChannelSet()
  int getSize()
  void addChannel(OutChannel c)
  OutChannel getChannel(int i)
  void send(Message m)
}
```

An object of this type represents a set of output channels. The default constructor creates an empty set. The other methods are described in the following subsection.

`Message`: Channel Messages

```
class Messaage
{
  String getText()
}
```

An object of (a subtype of) this type can be sent respectively received via a channel. The method `getText()` returns a string (default: "`(no information)`") that is displayed by the visualizer in a pop-up window whenever the mouse pointer is moved over the visual representation of a channel which contains this message. The programmer may override this method to exhibit the content of a message.

`GlobalAssertion`: Network Conditions

```
abstract class GlobalAssertion
{
  abstract boolean assert(Program program[])
  String getText()
  static Message[] getMessages(InChannel c)
  static Message[] getMessages(OutChannel c)
}
```

An object a of this type is passed to an `assert` call (see Subsection 5.5). The function `a.assert(p)` is then invoked on the array p of programs associated to the nodes of the network (in the order in which the nodes were created). If this function returns false, execution is aborted with the error message returned by `a.getText()`; otherwise execution is continued.

When `a.assert(p)` is called, it can rely on the fact that each program `p[i]` is either in the initial state, has terminated execution, is blocked on a communication operation *before* the corresponding message was sent respectively received (see Subsection 5.3), or has explicitly interrupted execution by a call of `yield` (see Subsection 5.5).

```
ms = getMessages(c)
  Message[] ms
  InChannel c
ms = getMessages(c)
  Message[] ms
  OutChannel c
```

This function may be used by an assertion to determine the contents of a channel c. Its return value is the array of messages contained in c in the order in which they were sent.

5.2 Node Interface

`in`: The Input Interface

```
s = in()
  InChannelSet s
c = in(i)
  InChannel c
  int i
```

`in()` returns the set of input channels entering the node on which the program is executed.

`in(i)` $(0 \leq i < s.\text{getSize}())$ returns the input channel with index i entering the node on which the program is ececuted.

`out`: The Output Interface

```
s = out()
  OutChannelSet s
c = out(i)
  OutChannel c
  int i
```

`out()` returns the set of output channels leaving the node on which the program is executed.

`out(i)` $(0 \leq i < s.\text{getSize}())$ returns the output channel with index i leaving the node on which the program is ececuted.

5.3 Basic Communication

`send`: Sending a Message

```
c.send(m)
  OutChannel c
  Message m
```

```
s.send(m)
  OutChannelSet s
  Message m
```

The first variant sends a message m to the output channel c; if the receiver node linked to c is blocked, this node is released.

The second variant sends m to all output channels in set s.

In both variants, the sender node is not blocked even if the receiver node linked to the corresponding channel is not ready to receive m. Consequently, c may hold an arbitrary number of messages.

`receive`: Receiving a Mesage

```
m = c.receive()
  Message m
  InChannel c
m = c.receive(t)
  Message m
  InChannel c
  int t;
```

The first variant returns a message from channel c; if c is empty, the current node is blocked until c becomes non-empty.

The second variant behaves like the first one except for the fact that the current node is blocked for at most t time units (see Subsection 5.5). If then c is still empty, `null` is returned.

Which message is returned from c depends on the *selector* associated to c; by default, this is the oldest message held by c (see Subsection 4.1).

5.4 Channel Sets

`getSize`: Determining the Size of a Channel Set

```
i = s.getSize()
  int i
  InChannelSet s
i = s.getSize()
  int i
  OutChannelSet s
```

This function returns the number of channels in set s.

`addChannel`: Adding a Channel to a Set

```
s.addChannel(c)
  InChannelSet s
  InChannel c
s.addChannel(c)
  OutChannelSet s
  OutChannel c
```

This function adds a channel c to a set s; c gets index i where $i = c$.getSize() (before the channel was added).

`getChannel`: Retrieving a Channel from a Set

```
c = s.getChannel(i)
  InChannel c
  InChannelSet s
  int i
c = s.getChannel(i)
  OutChannel c
  OutChannelSet s
  int i
```

The method returns the channel with index i ($0 \leq i < s$.getSize()) from set s.

`select`: Selecting a Channel from a Set

```
i = s.select()
i = s.select(t)
  int i
  InChannelSet s
  int t
```

The first variant returns the index of a non-empty channel in the (non-empty) set s; if all channels are empty, the current node is blocked until a channel becomes non-empty. This index may be used to reference the corresponding channel in the set (see Subsection 5.4).

The second variant behaves like the first one except for the fact that the current node is blocked for at most t time units (see Subsection 5.5). If then all channels are still empty, -1 is returned.

5.5 Execution Control

`assert`: Asserting Global Conditions

```
assert(a)
  GlobalAssertion a
```

This function checks whether the global assertion a (see Subsection 5.1) holds for all nodes of the network. If the assertion fails, execution of the network is terminated with an error message; otherwise execution is continued.

`getTime`: Determining the Node Time

```
i = getTime()
  int i
```

This function returns the local time of the current node. Time is guaranteed to advance by each invocation of `send`, `receive`, and `sleep`.

`sleep`: Letting Time Pass

```
sleep(i)
  int i
```

This function disables execution of the current node for *i* time units (see Subsection 5.5).

`yield`: Invoking the Scheduler

```
yield()
```

This function deschedules the current node, i.e., it gives another node the chance for execution. `yield` is implicitly called by each invocation of `send`, `receive`, and `select`. Apart from these operations, node execution is not interrupted. An invocation of `yield` *may* advance the local node time (see Subsection 5.5).

`interrupt`: Interrupting Network Execution

```
interrupt()
```

If the network is not executed in visual mode, this function does nothing. Otherwise, the execution of the network is interrupted until continued by the user.

`exit`: Terminating Network Execution

```
exit(i)
  int i
```

This function terminates the execution of the whole network with return code `i`.

References

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

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

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

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

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

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

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

A Proving Assertions

The assertion presented in Subsection 2.3 can (and should!) be also formally verified. In this appendix, we sketch a strategy how to prove properties of algorithms implemented with the facility of the DAJ toolkit. This strategy is based on the concept of Temporal Logic [7] but we deliberately avoid its formalism and sketch the ideas in natural language instead.

We focus on the proof of safety properties (properties that must *always* hold); at the end of this section we will shortly discuss liveness properties (properties that must *eventually* hold). Appendix B describes the design and verification of a more interesting distributed algorithm.

A.1 State Transitions

For the purpose of this subsection, we rewrite the method `main` as follows:

```
public void main()
{
  int i;
  int state = 0;
  while (state != 8) {
    switch (state) {
      case 0: {
```

```
          if (number = 0)
            state = 1;
          else
            state = 3;
        }
        case 1: {
          out(0).send(new Msg(0));
          state = 2;
        }
        case 2: {
          out(1).send(new Msg(0))
          sent = true;
          state = 3;
        }
        case 3: {
          i = 0;
          state = 4;
        }
        case 4: {
          if (i < 5)
            state = 5;
          else
            state = 8;
        }
        case 5: {
          index = in().select();
          state = 6;
        }
        case 6: {
          msg = in(index).receive();
          state = 7;
        }
        case 7: {
          out(index).send(msg);
          msg = null;
          i = i+1;
          state = 4;
        }
      }
    }
  }
```

This program implements the same behavior as the original one presented in the previous subsection by a sequence of state transitions such that the only interaction with other nodes (`send`, `receive`, `select`) occurs as the first statement of some transition. A node program may be interrupted only immediately before such a communication operation takes place. Network execution can be therefore considered as an interleaving of such transitions from the programs of different nodes.

A.2 The Invariant

To establish the truth of a global invariant, it thus suffices to prove

$$A \Rightarrow I$$
$$I \Rightarrow_{\circ} I$$

where A describes the initial state of the network (immediately before the transition loop is entered) and the implication $I \Rightarrow_{\circ} I$ denotes the invariance of I on every state transition (i.e., if I holds immediately before the transition, it must also hold after the transition).

The initial condition A of our network is defined as

$$\begin{array}{l} A :\Leftrightarrow \\ \quad \forall j \\ \quad\quad (out_j(0).\text{length} = 0,\ out_j(1).\text{length} = 0, \\ \quad\quad \neg sent_j,\ msg_j = \text{null},\ state_j = 0) \end{array}$$

i.e., all channels are empty, the local variables are initialized as described by the program constructor, and programs start with initial state 0.

We want to prove invariant I defined as

$$\begin{array}{l} I :\Leftrightarrow sent_0 \Rightarrow C(2) \\ C(c) :\Leftrightarrow c = \sum_j\ (out_j(0).\text{length} + out_j(1).\text{length} + n_j) \\ \quad\quad\quad \text{where} \\ \quad\quad\quad\quad n_j = 0, \text{ if } msg_j = \text{null} \\ \quad\quad\quad\quad n_j = 1, \text{ else} \end{array}$$

which is exactly the assertion implemented in the previous section. A subscripted variable v_j denotes the content of variable `v` in program `j` (j is represented in the program by the variable `number`). $out_j(i)$ denotes the array of messages `out(i).getMessages()` in program j.

For the purpose of verification, we need a couple of auxiliary propositions that describe the state at the beginning of each transition:

$$\begin{array}{l} S_j^0 :\Leftrightarrow state_j = 0 \Rightarrow \\ \quad msg_j = \text{null}, j = 0 \Rightarrow (C(0), \neg sent_0). \\ S_j^1 :\Leftrightarrow state_j = 1 \Rightarrow \\ \quad msg_j = \text{null}, j = 0,\ C(0),\ \neg sent_0. \\ S_j^2 :\Leftrightarrow state_j = 2 \Rightarrow \\ \quad msg_j = \text{null}, j = 0,\ C(1),\ \neg sent_0, \\ \quad out_j(1).\text{length} = 0. \\ S_j^3 :\Leftrightarrow state_j = 3 \Rightarrow \\ \quad msg_j = \text{null}. \\ S_j^4 :\Leftrightarrow state_j = 4 \Rightarrow \\ \quad msg_j = \text{null}. \end{array}$$

S_j^5 $:\Leftrightarrow$ $state_j = 5 \Rightarrow$
$msg_j =$ null, $i_j < 5$.
S_j^6 $:\Leftrightarrow$ $state_j = 6 \Rightarrow$
$msg_j =$ null, $i_j < 5$,
$index_j \in \{0, 1\}$, $in_j(index_j)$.length > 0.
S_j^7 $:\Leftrightarrow$ $state_j = 7 \Rightarrow$
$msg_j \neq$ null, $i_j < 5$,
$index_j \in \{0, 1\}$.
S_j^8 $:\Leftrightarrow$ $state_j = 8 \Rightarrow$
$msg_j =$ null.

We will prove a generalized invariant I' defined as

$$I' :\Leftrightarrow I, \forall j, k\ (S_j^k)$$

from which I trivially follows.

A.3 The Proof

We are now going to prove

$$A \Rightarrow I'$$
$$I' \Rightarrow_\circ I'$$

For proving $A \Rightarrow I'$, it suffices to prove $A \Rightarrow I$ and $A \Rightarrow \forall j\ (S_j^0)$ (both of which are direct consequences of A).

For proving $I' \Rightarrow_\circ I'$, we have to establish the truth of I after the transition and to show S_j^k; it suffices to consider only those j, k where

1. j is the index of the current node and k denotes a potential successor state of the transition, or
2. j is the index of another node and S_j^k is potentially affected by the transition.

In our program, the second case is only relevant for S_j^0, S_j^1, S_j^2 (because they contain propositions about *all* nodes) and S_j^6 (because it contains a proposition about the state of the node interface in_j which may be changed by some `send` operation).

We proceed by case distinction on the current state of node j, i.e., on the value of $state_j$:

- Assume $state_j = 0$: I is true after the transition, because $sent_0$ remains false. Since the transition does not involve a `send` or an assigment operation, nodes other than j are not affected. There are two potential successor states; it thus suffices to prove that the following conditions hold after the transition:

- $state_j = 1 \Rightarrow S_j^1$: S_j^1 holds because the premise implies that the "then" path of the conditional was taken and thus $j = 0$ holds.
- $state_j = 3 \Rightarrow S_j^3$: S_j^3 follows from S_j^0 and the code.

- Assume $state_j = 1$: I is true after the transition, because $sent_0$ remains false. From S_j^2 we know $j = 0$ which implies that $S_{j'}^0$, $S_{j'}^1$, $S_{j'}^2$ remain true for $j' \neq j$. The only effect of the the `send` operation to some node $j' \neq j$ might be the invalidation of $S_{j'}^6$. However, the statement

  ```
  c.send(m)
  ```

  satisfies the transition rule

  $$c = q \Rightarrow_\circ c = q{:}m$$

  (i.e., it appends one element m to the message queue q of channel c), thus $in_{j'}(i)$.length > 0 cannot become false.

  It remains to be shown that $state_j = 2 \Rightarrow S_j^2$ after the transition. From $C(0)$ at the beginning of the transition, we know $out_j(1)$.length $= 0$, which remains true. Furthermore, by above transition rule, we can easily deduce $out_j(0)$.length $= 1$ and thus $C(1)$ after the transition.

- Assume $state_j = 2$: we have to show that I remains true. By same reasoning as for the previous case, we get $out_j(1)$.length $= 1$ after the transition. Together with $C(1)$ and $out_j(1)$.length $= 1$ at the beginning of the transition, we can deduce $C(2)$ after the transition.

  By same reasoning as for the previous case, no $S_{j'}^k$ is affected by the transition. That S_j^3 holds after the transition follows directly from the code and from S_j^2.

- Assume $state_j = 3$: from the code, the truth of I is self-evident, no $S_{j'}^k$ is affected and by S_j^3 also S_j^4 is clear.

- Assume $state_j = 4$: it suffices to prove that the following conditions hold after the transition:
  - $state_j = 5 \Rightarrow S_j^5$: $i < 5$ holds because the premise implies that the "then" part of the conditonal was taken.
  - $state_j = 8 \Rightarrow S_j^8$: this is self-evident.

- Assume $state_j = 5$: the truth of I is clear and $index_j$ is not referenced in any $S_{j'}^k$ with $j' \neq j$. Thus it suffices to show that S_j^6 holds after the transition. The statement

  ```
  i = s.select()
  ```

  satisfies the transition rule

  $$s.\text{getSize}() > 0 \Rightarrow_\circ 0 \leq i < s.\text{getSize}(),\ s(i).\text{length} > 0$$

  and we know by network construction out.getSize() $= 2$. Thus, using S_j^5, the truth of S_j^6 is clear.

- Assume $state_j = 6$: first we show the truth of I after the transition. The statement

  ```
  m = c.receive()
  ```

  satisfies the transition rule

  $$c = n{:}q \Rightarrow_\circ m = n,\ c = q$$

  (for some non-null message n and message queue q). From this and S_j^6 and knowing out.getSize() $= 2$ it is clear that, for some $k \in \{0,\ 1\}$ and non-negative l,

  $$in_j(k).\text{length} = 1 + l$$

  holds before the transition, and that

  $$in_j(k).\text{length} = l,\ msg_j \neq \text{null}$$

  holds after the transition and that $k = index_j$ (both before and after the transition). From this and from the overall relationship between input and output channels

  $$\begin{aligned}&\textstyle\sum_j\ (in_j(0).\text{length} + in_j(1).\text{length}) = \\ &\textstyle\sum_j\ (out_j(0).\text{lenght} + out_j(1).\text{length})\end{aligned}$$

  it is clear that I holds after the transition.

  By above reasoning, it is also clear that S_j^7 holds after the transition.

  From S_j^6 and by above relationship between input and output channels we know before the transition $out_{j'}(p)$.length > 0 for some $j' \neq j$ and $p \in \{0, 1\}$ (by network construction). Thus $state_0 \neq 0$, $state_{j'} \neq 1$ (for all $j' \neq j$) and the transition cannot invalidate $S_{j'}^0$ and $S_{j'}^1$. $S_{j'}^2$ remains true because $\texttt{out}_{j'}(1)$.length() $= 0$ cannot be changed by a `receive` operation; furthermore I and $C(1)$ at the beginning of the transition implies by the code and above rule also $C(1)$ at the end of the transition. $S_{j'}^6$ remains true because the truth of $in_{j'}(index_{j'})$.length > 1 cannot be changed by a `receive` operation in j.

- Assume $state_j = 7$: the truth of I is clear from the fact that msg_j is non-null before the transition but null after the transition using the transition rule for `send` shown above.

  Similar reasoning as in the previous case also shows that $S_0^{j'}$, $S_{j'}^1$ and $S_{j'}^2$ remain true. $S_{j'}^6$ remains true because the truth of $in_{j'}(index_{j'})$.length $>$ 1 cannot be changed by a `send` operation.

Thus we have established the invariance of I'.

A.4 Termination

Since each program loops through a bounded number of iterations, it is clear that the execution of the network will not run forever. It is however less clear whether all node programs actually reach the termination state. In fact, this is *not* guaranteed: two network programs may terminate in a state where both messages are captured in the channels between them; the third node will thus wait forever on its input channels.

B Distributed Snapshots

B.1 Problem

We are going to design and to implement a distributed program that simulates the behavior of a banking system:

- every node of the network holds a money deposit, and
- every node may randomly withdraw money from its deposit and send it to some neighbor node which stores the money in its own deposit.

Let the *network value* be the sum of all money stored in some node or contained in some message of the network; by above operation this value remains constant but is unknown to every node.

The program is to determine the network value in every node without interrupting normal operation; when the value is known to every node, execution may terminate. An assertion should check whether the result is correct.

B.2 Algorithm

We implement the solution of the problem by an application of the *Chandy-Lamport Algorithm* for determining *consistent global snapshots* of a network[5]. In our case, this algorithm is applied as follows:

- Initially all nodes are in mode *RUNNING* exchanging value messages as described in the problem statement. At some time, node 0 switches to mode *SNAPPING* and sends out snap messages (see below); every node that receives a snap message, also switches to mode *SNAPPING*.
- When a node switches to mode *SNAPPING*, it determines the current value of its local deposit in a variable *snapValue* and sends snap messages to all output channels.
- Each node in mode *SNAPPING* continues normal operation. However the value of every message from every input channel is added to *snapValue* until a snap message arrives through that channel.

[5] K. Mani Chandy and Leslie Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

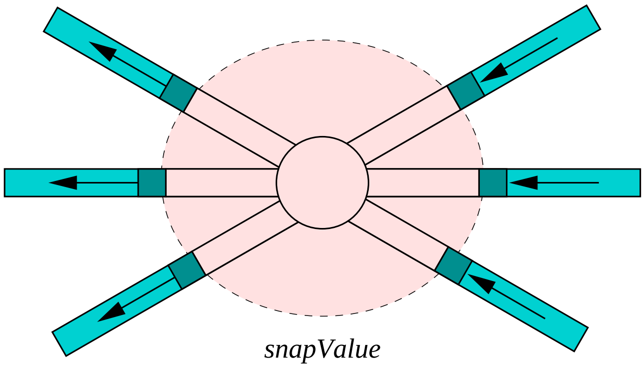

Figure 2: Distributed Snapshots: Basic Idea

- If snap messages have arrived through all input channels of a node, the current *snapValue* represents the node's part of the network value.

Figure 2 illustrates the basic idea of the algorithm: the stream of messages passing through a channel is split by the snap message into two parts; a node's *snapValue* eventually represents the value of the node after having processed all messages within the partition bounded by the snap messages of the incoming and outgoing channels. Since every message is contained in exactly one such partition, the sum of all *snapValue*s represents the total network value.

In order to broadcast a node's *snapValue* to every other node of the network, the following (simple-minded) strategy is applied:

- When a node has determined its final *snapValue*, it sends this value (together with its node number) to all neighbors and continues normal operation.
- A node that receives some other node's *snapValue* determines whether it has already seen this value. If not, this value is added to the node's local variable *totalValue* and the message is forwarded to all neighbor nodes.
- If a node has seen and forwarded the *snapValue* of every other node, it terminates execution.

The program thus ultimately terminates in a state where every node holds the same *totalValue* which represents the value of the network.

B.3 Program

Below you can find the full source code of the program implementing the algorithm sketched in the previous subsection. The algorithm is executed on a hexagon network as visualized in Figure 4.

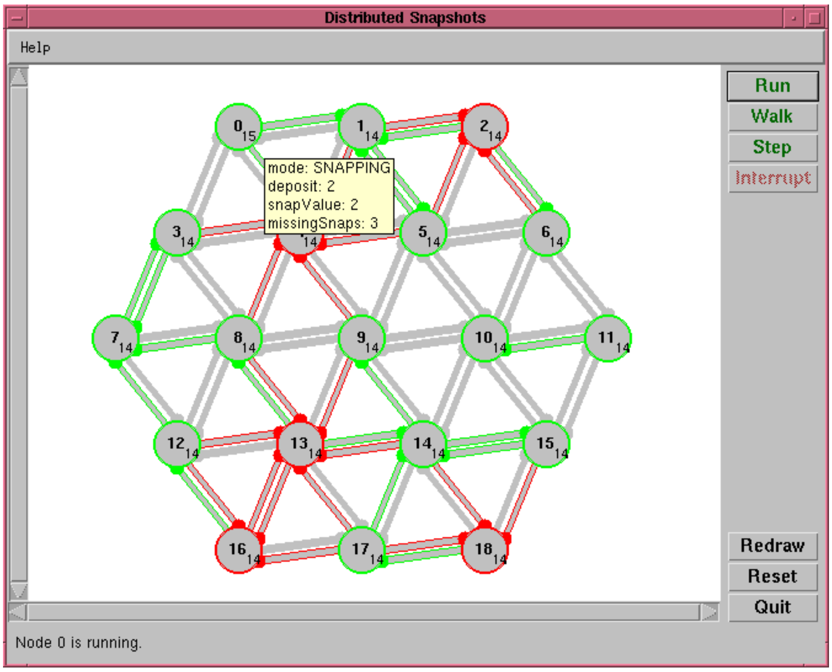

Figure 3: Distributed Snapshots: Visualization

```
// ------------------------------------------------------------------------
// $Id: snapshot.tex,v 1.3 1997/11/12 21:59:23 schreine Exp $
// the Chandy-Lamport algorithm for taking distributed snapshots
//
// (c) 1997, Wolfgang Schreiner <Wolfgang.Schreiner@risc.uni-linz.ac.at>
// http://www.risc.uni-linz.ac.at/software/daj
// ------------------------------------------------------------------------
import daj.*;
import java.util.*;

// --------------------------------------------------------------------------
//
// the application itself
//
// --------------------------------------------------------------------------
public class Main extends Application
{
  // ------------------------------------------------------------------------
  // main function of application
  // ------------------------------------------------------------------------
  public static void main(String[] args)
  {
    new Main().run();
  }

  // ------------------------------------------------------------------------
  // constructor for application
  // ------------------------------------------------------------------------
  public Main()
  {
```

```
    super("Distributed Snapshots", 580, 450);
  }

  // -------------------------------------------------------------------------
  // construction of a hexagon network (just for fun)
  // -------------------------------------------------------------------------
  public void construct()
  {
    // length of first row and half height of hexagon
    int w = 3;
    int h = 2;
    int size = (w+h-1)*(w+h)-w*(w-1)+(w+h);
    Node nodes[][] = new Node[w+h][2*h+1];

    // left upper corner and vertical/horizontal distance
    int x0 = 170;
    int y0 = 50;
    int x = 100;
    int y = 86;

    // random generator for initialization of deposit
    Random random = new Random();

    // create and place nodes
    int number = 0;
    for (int j = 0; j < h+1; j++)
      {
        for (int i = 0; i < w+j; i++)
          {
            int deposit = Math.abs(random.nextInt())%1000;
            nodes[i][j] =
              node(new Prog(size, number, deposit), String.valueOf(number),
                   x0+i*x-j*x/2, y0+j*y);
            number++;

          }
      }
    for (int j = h+1; j < 2*h+1; j++)
      {
        for (int i = 0; i < w+2*h-j; i++)
          {
            int deposit = Math.abs(random.nextInt())%1000;
            nodes[i][j] =
              node(new Prog(size, number, deposit), String.valueOf(number),
                   x0+i*x-(2*h-j)*x/2, y0+j*y);
            number++;
          }
      }

    // link nodes
    for (int j = 0; j < h+1; j++)
      {
        for (int i = 0; i < w+j; i++)
          {
            if (i < w+j-1) link(nodes[i][j], nodes[i+1][j]);
            if (i > 0) link(nodes[i][j], nodes[i-1][j]);
            if (j < h)
              {
                link(nodes[i][j], nodes[i][j+1]);
                link(nodes[i][j], nodes[i+1][j+1]);
              }
            else
```

```
              {
                if (i < w+j-1) link(nodes[i][j], nodes[i][j+1]);
                if (i> 0) link(nodes[i][j], nodes[i-1][j+1]);
              }
            if (j > 0)
              {
                if (i < w+j-1) link(nodes[i][j], nodes[i][j-1]);
                if (i > 0) link(nodes[i][j], nodes[i-1][j-1]);
              }
          }
      }
    for (int j = h+1; j < 2*h+1; j++)
      {
        for (int i = 0; i < w+2*h-j; i++)
          {
            if (i < w+2*h-j-1) link(nodes[i][j], nodes[i+1][j]);
            if (i > 0) link(nodes[i][j], nodes[i-1][j]);
            if (j < 2*h)
              {
                if (i > 0) link(nodes[i][j], nodes[i-1][j+1]);
                if (i < w+2*h-j-1) link(nodes[i][j], nodes[i][j+1]);
              }
            if (j > 0)
              {
                link(nodes[i][j], nodes[i][j-1]);
                link(nodes[i][j], nodes[i+1][j-1]);
              }
          }
      }
  }

  // -------------------------------------------------------------------------
  // informative message
  // -------------------------------------------------------------------------
  public String getText()
  {
    return "Distributed Snapshots\n \n" +
      "The Chandy-Lamport algorithm\n" +
      "for taking consistent global snapshots\n" +
      "of a running network.";
  }
}

// ---------------------------------------------------------------------------
//
// a program class
//
// ---------------------------------------------------------------------------
class Prog extends Program
{
  // state variables
  public int size;        // network size
  public int number;      // current node number
  public int deposit;     // current deposit
  public Message message; // message currently being processed
  public int index;       // index of channel from which message was received
  public int mode;        // current execution mode

  // snapped local state
  public int snapValue;      // current snap value
  public boolean snapped[];  // snap state of input channels
  public int missingSnaps;   // number of missing snaps of input channels
```

```
// snapped total state
public int totalValue;     // current network value
public boolean done[];     // true iff node has reported its snap value
public int missingValues;  // number of outstanding node reports

// execution modes
private final int RUNNING = 0;      // initial state
private final int SNAPPING = 1;     // constructing snapshot
private final int BROADCASTING = 2; // broadcasting result
private final int TERMINATED = 3;   // terminal state

// ------------------------------------------------------------------------
// construct program in network of size `s` at node `n` with deposit `d`
// ------------------------------------------------------------------------
public Prog(int s, int n, int d)
{
  size = s;
  number = n;
  deposit = d;
  message = null;
  mode = RUNNING;
}

// ------------------------------------------------------------------------
// return random integer `r` with 0 <= `r` < `n`
// ------------------------------------------------------------------------
private Random rand = new Random();
private int random(int n)
{
  return Math.abs(rand.nextInt())%n;
}

// ------------------------------------------------------------------------
// go into snapping mode
// ------------------------------------------------------------------------
public void snap()
{
  // go into snap mode, store local deposit, reset total value
  mode = SNAPPING;
  snapValue = deposit;
  totalValue = 0;

  // initialize snapping state of channels
  int s = in().getSize();
  snapped = new boolean[s];
  for (int i = 0; i < s; i++)
    snapped[i] = false;
  missingSnaps = s;

  // initialize report state of nodes
  done = new boolean[size];
  for (int i = 0; i < size; i++)
    done[i] = false;
  missingValues = size;
}

// ------------------------------------------------------------------------
// collect `value` from `sender`
// ------------------------------------------------------------------------
public void collect(int sender, int value)
{
```

```
    totalValue += value;
    done[sender] = true;
    missingValues--;
    if (missingValues == 0)
      mode = TERMINATED;
  }

  // -----------------------------------------------------------------------
  // main function of program
  // -----------------------------------------------------------------------
  public void main()
  {
    while (mode != TERMINATED)
      {
        // node 0 triggers snapping at its own will
        if (number == 0 && mode == RUNNING && random(10) == 0)
          {
            snap();
            interrupt();
            out().send(new SnapMessage());
          }

        // wait some time for message
        index = in().select(random(5));
        if (index == -1 && deposit > 0)
          {
            // -------------------------------------------------------------
            // send some money from deposit to some neighbor node
            // -------------------------------------------------------------
            int value = random(deposit);
            deposit -= value;
            message = new ValueMessage(value);
            index = random(out().getSize());
            out(index).send(message);
            message = null;
            continue;
          }

        // message was received
        message = in(index).receive();

        // -------------------------------------------------------------
        // value message was received
        // -------------------------------------------------------------
        if (message instanceof ValueMessage)
          {
            ValueMessage valueMessage = (ValueMessage)message;
            int value = valueMessage.getValue();
            deposit += value;

            // snap message
            if (mode == SNAPPING && !snapped[index])
              {
                snapValue += value;
              }
          }

        // -------------------------------------------------------------
        // snap message was received
        // -------------------------------------------------------------
        else if (message instanceof SnapMessage)
          {
```

```
          SnapMessage snapMessage = (SnapMessage)message;
          if (mode == RUNNING)
            {
              snap();
              out().send(snapMessage);
            }
          if (!snapped[index])
            {
              snapped[index] = true;
              missingSnaps--;
              if (missingSnaps == 0)
                {
                  mode = BROADCASTING;
                  collect(number, snapValue);
                  if (number == 0) interrupt();
                  out().send(new ResultMessage(number, snapValue));
                }
            }
        }

      // -----------------------------------------------------------------
      // result message was received
      // -----------------------------------------------------------------
      else if (message instanceof ResultMessage)
        {
          ResultMessage resultMessage = (ResultMessage)message;
          int sender = resultMessage.getSender();
          if (!done[sender])
            {
              collect(sender, resultMessage.getValue());
              out().send(resultMessage);
            }
        }

      // message was handled
      message = null;
    }
  // assert that total amount of money equals the determined value
  assert(new TotalValue(totalValue));
}

// ---------------------------------------------------------------------------
// called for display of program state
// ---------------------------------------------------------------------------
public String getText()
{
  String messageString;
  if (message == null)
    messageString = "(null)";
  else
    messageString = message.getText();
  String modeString;
  switch (mode)
    {
    case RUNNING:
      {
        modeString = "RUNNING";
        break;
      }
    case SNAPPING:
      {
        modeString = "SNAPPING";
```

```
          break;
        }
      case BROADCASTING:
        {
          modeString = "BROADCASTING";
          break;
        }
      case TERMINATED:
        {
          modeString = "TERMINATED";
          break;
        }
      default:
        {
          modeString = "(unknown)";
          break;
        }
      }
    if (message == null)
      messageString = "(null)";
    else
      messageString = message.getText();

    if (mode == RUNNING)
      return
        "mode: " + modeString +
        "\ndeposit: " + String.valueOf(deposit);
    else if (mode == SNAPPING)
      return
        "mode: " + modeString +
        "\ndeposit: " + String.valueOf(deposit) +
        "\nsnapValue: "  + String.valueOf(snapValue) +
        "\nmissingSnaps: "  + String.valueOf(missingSnaps);
    else if (mode == BROADCASTING)
      return
        "mode: " + modeString +
        "\ndeposit: " + String.valueOf(deposit) +
        "\nsnapValue: "  + String.valueOf(snapValue) +
        "\ntotalValue: " + String.valueOf(totalValue) +
        "\nmissingValues: "  + String.valueOf(missingValues);
    else
      return
        "mode: " + modeString +
        "\ndeposit: " + String.valueOf(deposit) +
        "\nsnapValue: "  + String.valueOf(snapValue) +
        "\ntotalValue: " + String.valueOf(totalValue);
  }
}

// ---------------------------------------------------------------------------
//
// the message classes
//
// ---------------------------------------------------------------------------


// ---------------------------------------------------------------------------
// carries `value`
// ---------------------------------------------------------------------------
class ValueMessage extends Message
{
  private int value;
```

```
  public ValueMessage(int v)
  {
    value = v;
  }
  public int getValue()
  {
    return value;
  }
  public String getText()
  {
    return "VALUE[" + String.valueOf(value) + "]";
  }
}

// --------------------------------------------------------------------------
// carries no information
// --------------------------------------------------------------------------
class SnapMessage extends Message
{
  public String getText()
  {
    return "SNAP";
  }
}

// --------------------------------------------------------------------------
// carries `value` of `sender`
// --------------------------------------------------------------------------
class ResultMessage extends Message
{
  private int sender;
  private int value;
  public ResultMessage(int s, int v)
  {
    sender = s;
    value = v;
  }
  public int getSender()
  {
    return sender;
  }
  public int getValue()
  {
    return value;
  }
  public String getText()
  {
    return "RESULT[" + String.valueOf(sender) + ", " +
      String.valueOf(value) + "]";
  }
}

// --------------------------------------------------------------------------
//
// global assertion stating that total values of network equals `value`
//
// --------------------------------------------------------------------------
class TotalValue extends GlobalAssertion
{
  int value; // expected value
  int sum;   // counted value
  public TotalValue(int v)
```

```
  {
    value = v;
  }
  public String getText()
  {
    return "invalid value: " + String.valueOf(sum) +
      "(" + String.valueOf(value) + " expected)";
  }

  // -------------------------------------------------------------------------
  // add values of all messages in `msg`
  // -------------------------------------------------------------------------
  private int add(Message msg[])
  {
    int s = 0;
    for (int i = 0; i < msg.length; i++)
      {
        if (msg[i] instanceof ValueMessage)
          {
            s += ((ValueMessage)msg[i]).getValue();
          }
      }
    return s;
  }

  // -------------------------------------------------------------------------
  // check whether total value of network equals `value`
  // -------------------------------------------------------------------------
  public boolean assert(Program prog[])
  {
    sum = 0;
    for (int j = 0; j < prog.length; j++)
      {
        Prog program = (Prog)prog[j];
        // value deposited in node
        sum += program.deposit;
        // value in output channels
        int n = program.out().getSize();
        for (int i = 0; i < n; i++)
          {
            Message msg[] = getMessages(program.out(i));
            sum += add(msg);
          }
        // value in pending message
        if (program.message != null && program.message instanceof ValueMessage)
          {
            sum += ((ValueMessage)program.message).getValue();
          }
      }
    return sum == value;
  }
}
```

B.4 State Program

For the purpose of verification, we assume that the initializations

```
message = null;
mode = RUNNING;
```

have taken place and rewrite the program as shown below to make the interactions with other nodes explicit by state changes:

```
state = LOOP;
while (state != EXIT)
{
  switch (state)
    {
    case LOOP:
      {
        if (mode == TERMINATED)
          state = EXIT;
        else if (number == 0 && mode == RUNNING && random(10) == 0)
          {
            snap();
            state = TRIGGER;
          }
        else
          state = SELECT;
      }
    case TRIGGER:
      {
        out().send(new SnapMessage());
        state = SELECT;
      }
    case SELECT:
      {
        index = in().select(random(5));
        if (index == -1 && deposit > 0)
          {
            int value = random(deposit);
            deposit -= value;
            message = new ValueMessage(value);
            index = random(out().getSize());
            state = DEPOSIT;
          }
        else
          state = RECEIVE;
      }
    case DEPOSIT:
      {
        out(index).send(message);
        message = null;
        state = LOOP;
      }
    case RECEIVE:
      {
        message = in(index).receive();
        if (message instanceof ValueMessage)
          {
            ValueMessage valueMessage = (ValueMessage)message;
            int value = valueMessage.getValue();
            deposit += value;
            if (mode == SNAPPING && !snapped[index])
              snapValue += value;
            message = null;
            state = LOOP;
          }
        else if (message instanceof SnapMessage)
          {
            if (mode != SNAPPING)
              {
                snap();
```

```
            state = SNAP;
          }
        else
          state = SNAP2;
      }
    else if (message instanceof ResultMessage)
      {
        ResultMessage resultMessage = (ResultMessage)message;
        int sender = resultMessage.getSender();
        if (!done[sender])
          {
            collect(sender, resultMessage.getValue());
            state = RESULT;
          }
        else
          {
            message = null;
            state = LOOP;
          }
      }
  }
case SNAP:
  {
    out().send(message);
    state = SNAP2;
  }
case SNAP2:
  {
    if (!snapped[index])
      {
        snapped[index] = true;
        missingSnaps--;
        if (missingSnaps == 0)
          {
            mode = BROADCASTING;
            collect(number, snapValue);
            state = SNAP3;
          }
        else
          {
            message = null;
            state = LOOP;
          }
      }
    else
      {
        message = null;
        state = LOOP;
      }
  }
case SNAP3:
  {
    out().send(new ResultMessage(number, snapValue));
    message = null;
    state = LOOP;
  }
case RESULT:
  {
    out().send(message);
    message = null;
    state = LOOP;
  }
```

```
    }
}
```

Above program pretends that sending a message to a *set* of output channels is an atomic operation. This is in fact *not* true in DAJ; the sender node may be interrupted while only some part of the output channels has received the message. This property can be modeled correctly in above framework by an iteration of send operations of single channels. However, the proof described in the next section will not involve any property that may be invalidated by the fact that messages are not put simultaneously into all output channels (only snap and result messages are broadcast); thus we stick to the simplified model.

B.5 Verification

We will prove the correctness of the program shown above; this is *not* a proof of the Chandy-Lamport algorithm that solves a much more general problem [6]. However, by focusing on our application, the proof becomes more concrete and its relationship to the program more direct.

In this section, we wil use the following typed variables:

m	a message
c	a channel, identified with the array of messages in it
i, j	indices of network nodes
k	an index of a message in c, $0 \leq k < c.\text{length}$
h	an index of a channel in a set s, $0 \leq k < s.\text{getSize}()$

Furthermore, we will denote by $d_i \geq 0$ the initial value of $deposit_i$ and by n the number of nodes in the network.

Correctness Theorem

The following definitions capture the measures we are interested in:

Definition 1 (Network Value) *The network value $value_{net}$ is the sum of all node deposits plus the sum of the values of all messages being processed in some node plus all channel values:*

$$value_{net} := \sum_i (deposit_i + mvalue(message_i)) + \sum_c cvalue(c).$$

Definition 2 (Channel Value) *The channel value cvalue(c) of a channel c is the sum of all message values in c:*

$$cvalue(c) := \sum_k mvalue(c[k])$$

Definition 3 (Message Value) *The message value mvalue(m) of a message m is the value carried by m, if m is a value message, and 0, otherwise:*

$$mvalue(m) := \begin{cases} m.getValue() & if\ m\ instanceOf\ ValueMessage \\ 0 & else \end{cases}$$

The correctness theorem for our program can be then expressed as follows:

Theorem 1 (Correctness) *When a node terminates, its totalValue equals the network value:*

$$mode_i = TERMINATED \Rightarrow totalValue_i = value_{net}$$

Invariants

In order to formulate the program invariants from which the correctness theorem can be proved, we need to refer to the all the messages that have been ever sent through a channel:

Definition 4 (Channel Trace) *The trace of channel c is a triple (C, $inpos_c$, $outpos_c$) such that C is an array that holds the stream of all messages that have been sent through c, the next message that will be read by the receiver is stored in C[$inpos_c$], and the next message sent by the sender will be written into C[$outpos_c$] with $inpos_c \leq outpos_c$.*

Apparently, a channel c is empty if $inpos_c = outpos_c$.

We may think of a channel trace as an additional data structure where $inpos_c$ and $outpos_c$ are initialized as zero; the data structure is updated by every operation c.send(v) as follows:

```
C[outpos_c] = v;
outpos_c++
```

In addition the operation `v = c.receive()` has the effect

```
inpos_c++
```

We may then introduce a derived notion:

Definition 5 (Trace Value) *The trace value tvalue(c, from, to) of a channel c in interval [from,to[is the sum of the values of the messages in the trace of c between (including) 'from' and (not including) 'to':*

$$tvalue(c, from, to) := \sum_{from \leq k < to} mvalue(C[k])$$

This notion can be used to express the value of a network as follows:

Theorem 2 (Network Value) *The value of a network equals the sum of all initial node deposits plus the sum of the values of all messages being processed by some node plus the sum of the values of the messages received by each node minus the sum of the values of the messages left by each node:*

$$\begin{aligned}&\mathit{value}_{net} = \\ &\quad \textstyle\sum_i \ (\mathrm{d}_i + \mathit{mvalue}(\mathrm{message}_i)) + \\ &\quad \textstyle\sum_c \ \mathit{tvalue}(c,\ \mathrm{inpos}_c,\ \mathrm{outpos}_c))\end{aligned}$$

Proof: The result follows from the fact that the network value remains constant by the operation of the program; the proof of this proceeds analogously to the proof in Section A.

We now characterize the behavior of the program by the following set of propositions which we have to prove as invariants:

Definition 6 (Invariants) *For every node i, we define the following propositions:*

1. ***D**$_i$ **(Deposit)**: the node deposit is non-negative and equals the initial deposit value plus (potentially) the value of the message just received plus the current trace value of all input channels minus the current trace value of all output channels:*

$$\begin{aligned}&\mathit{deposit}_i \geq 0, \\ &\mathit{deposit}_i = \mathrm{d}_i \\ &\quad + \textstyle\sum_{c \in in_i} \ \mathit{tvalue}(c,\ 0,\ \mathrm{inpos}_c) \\ &\quad - \textstyle\sum_{c \in out_i} \ \mathit{tvalue}(c,\ 0,\ \mathrm{outpos}_c)\end{aligned}$$

2. ***M**$_i$ **(Snap Messages)**: if node i is in mode RUNNING, it has not yet sent a snap message to any of its output channels and it has not yet received a snap message from any of its input channels:*

$$\begin{aligned}&\mathrm{mode}_i = \mathit{RUNNING} \Rightarrow \\ &\quad \forall c \in \mathrm{out}_i \ \forall k \ (\neg C[k] \ \mathit{instanceOf\ SnapMessage}) \\ &\quad \forall c \in \mathrm{in}_i \ \forall k < \mathrm{inpos}_c \ (\neg C[k] \ \mathit{instanceOf\ SnapMessage})\end{aligned}$$

If the node is not in mode RUNNING any more, it has sent exactly one snap message to every of its output channels and it has received exactly one snap message in every input channel where the snapped *flag is set:*

$$\begin{aligned}&\mathrm{mode}_i \neq \mathit{RUNNING} \Rightarrow \\ &\quad \forall c \in \mathrm{out}_i \ \exists! \ \mathrm{snap} \ (C[\mathrm{snap}] \ \mathit{instanceOf\ SnapMessage}) \\ &\quad \forall c \in \mathrm{in}_i \ \forall h \ ((c = \mathrm{in}_i(h),\ \mathrm{snapped}_i[h]) \Rightarrow \\ &\qquad \exists! \ \mathrm{snap} < \mathrm{inpos}_c \ (C[\mathrm{snap}] \ \mathit{instanceOf\ SnapMessage}))\end{aligned}$$

If a node is not in mode RUNNING and c is an input or output channel of this node, we denote by snap_c *the unique position of the snap message in the trace of c.*

3. ***S**$_i$ **(Snapping)**: if node i is not in mode RUNNING any more, the snap value equals the deposit plus the trace value of all output channels from the position of their snap messages sent by the node minus the trace value of all input channels from the position of their snap messages received by the node (if such a snap message was received);* missingSnaps *denotes the number of outstanding snap messages:*

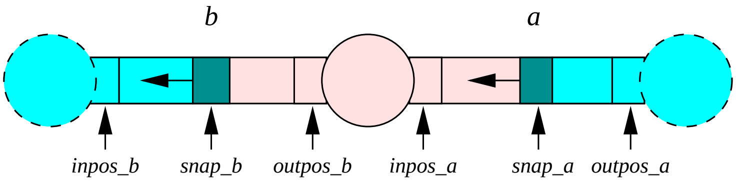

Figure 4: Distributed Snapshots: Proof

$$\begin{aligned}
&\mathit{mode}_i \neq \mathit{RUNNING} \Rightarrow \\
&\quad \mathit{snapValue}_i = \mathit{deposit}_i \\
&\qquad + \textstyle\sum_{c\in\ out_i} \mathit{tvalue}(c,\ \mathit{snap}_c,\ \mathit{outpos}_c) \\
&\qquad - \textstyle\sum_{c\in\ in_i, c\ =\ in_i(h),\ snapped_i[h]} \mathit{tvalue}(c,\ \mathit{snap}_c,\ \mathit{inpos}_c), \\
&\quad \mathit{missingSnaps} = \textstyle\sum_{h,\neg snapped[h]} 1
\end{aligned}$$

4. ***B**$_i$ **(Broadcasting)**: if node i is in one of the modes BROADCASTING or TERMINATED, then its totalValue equals the sum of all received snapValues and missingValues denotes the number of all not yet received snapValues:*

$$\begin{aligned}
&\mathit{mode}_i \in \{\mathit{BROADCASTING},\ \mathit{TERMINATED}\} \Rightarrow \\
&\quad \forall_{j, done[j]}\ \mathit{mode}_j \in \{\mathit{BROADCASTING},\ \mathit{TERMINATED}\}, \\
&\quad \forall c \in in_i\ \forall h\ (c = in(h) \Rightarrow \mathit{snapped}[h]) \\
&\quad \mathit{totalValue} = \textstyle\sum_{j, done[j]} \mathit{snapValue}_i \\
&\quad \mathit{missingValues} = \textstyle\sum_{j,\neg done[j]} 1
\end{aligned}$$

5. ***T**$_i$ **(Terminated)**: if the node is in mode TERMINATED, no snap values are missing any more:*

$$\begin{aligned}
&\mathit{mode}_i = \mathit{TERMINATED} \Rightarrow \\
&\quad \mathit{missingValues}_i = 0.
\end{aligned}$$

Let I be the conjunction of all propositions above (for all i). Assuming that I is a program invariant, we can proof Theorem 1 as follows (see also Figure 4):

Proof of Correctness Theorem: When node i is in mode TERMINATED, we know by T_i that $\mathit{missingValues}_i = 0$. By B_i, this means that $\mathit{totalValue}_i = \sum_j \mathit{snapValue}_j$, that every node j is in one of the modes BROADCASTING or TERMINATED, and that $\mathit{snapped}[k]$ for every index k of an input channel. Together with S_j for every node j, we thus have

$$\begin{aligned}
&\textstyle\sum_j \mathit{snapValue}_j = \sum_j \\
&\quad \mathit{deposit}_j \\
&\qquad + \textstyle\sum_{c\in out_j} \mathrm{tvalue}(c,\ \mathit{snap}_c,\ \mathit{outpos}_c) \\
&\qquad - \textstyle\sum_{c\in in_j} \mathrm{tvalue}(c,\ \mathit{snap}_c,\ \mathit{inpos}_c)
\end{aligned}$$

From S_j for every node j, we know for every channel c that $0 \leq \mathit{snap}_c < \mathit{inpos}_c \leq \mathit{outpos}_c$. Since every channel has exactly one sender and one receiver node, we then have

$$
\begin{aligned}
&\textstyle\sum_j \mathit{snapValue}_j \\
&= \textstyle\sum_j \mathit{deposit}_j \\
&\quad + \textstyle\sum_c \mathrm{tvalue}(c, \mathit{snap}_c, \mathit{outpos}_c) \\
&\quad - \textstyle\sum_c \mathrm{tvalue}(c, \mathit{snap}_c, \mathit{inpos}_c) \\
&= \textstyle\sum_j \mathit{deposit}_j + \sum_c \mathrm{tvalue}(c, \mathit{inpos}_c, \mathit{outpos}_c)
\end{aligned}
$$

By D_i, we know

$$\textstyle\sum_j \mathit{deposit}_j + \sum_c \mathrm{tvalue}(c, \mathit{inpos}_c, \mathit{outpos}_c) = \sum_j d_j$$

and, by Theorem 2,

$$\textstyle\sum_j d_j = \mathrm{value}_{\mathrm{net}}$$

which completes the proof.

Program Properties

For proving the invariance of I, we first define a property that holds after each invocation of `snap` in node i:

$$
\begin{aligned}
&\mathit{snap}_i :\Leftrightarrow \\
&\quad \mathit{mode}_i = \mathrm{SNAPPING}, \\
&\quad \mathit{snapValue}_i = \mathit{deposit}_i, \\
&\quad \mathit{totalValue}_i = 0, \\
&\quad \forall h\ \neg \mathit{snapped}_i[h], \\
&\quad \mathit{missingSnaps}_i = \mathit{in}_i().\mathrm{getSize}(), \\
&\quad \forall j\ \neg \mathit{done}_i[j], \\
&\quad \mathit{missingValues}_i = n.
\end{aligned}
$$

Then we are going to define the properties that characterize the various states of node i:

$$
\begin{aligned}
&S_i^{\mathrm{LOOP}} :\Leftrightarrow \mathit{state}_i = \mathrm{LOOP} \Rightarrow \\
&\quad \mathit{message}_i = \mathrm{null}. \\
&S_i^{\mathrm{TRIGGER}} :\Leftrightarrow \mathit{state}_i = \mathrm{TRIGGER} \Rightarrow \\
&\quad i = 0,\ \mathit{message}_i = \mathrm{null},\ \mathit{snap}_i. \\
&S_i^{\mathrm{SELECT}} :\Leftrightarrow \mathit{state}_i = \mathrm{SELECT} \Rightarrow \\
&\quad \mathit{mode}_i \neq \mathrm{TERMINATED}. \\
&S_i^{\mathrm{DEPOSIT}} :\Leftrightarrow \mathit{state}_i = \mathrm{DEPOSIT} \Rightarrow \\
&\quad \mathit{mode}_i \neq \mathrm{TERMINATED},\ 0 \leq \mathit{index}_i < \mathit{out}_i().\mathrm{getSize}(), \\
&\quad \mathit{message}_i \text{ instanceOf ValueMessage}. \\
&S_i^{\mathrm{RECEIVE}} :\Leftrightarrow \mathit{state}_i = \mathrm{RECEIVE} \Rightarrow \\
&\quad \mathit{mode}_i \neq \mathrm{TERMINATED},\ 0 \leq \mathit{index}_i < \mathit{in}_i().\mathrm{getSize}(), \\
&\quad \mathit{inpos}_{\mathit{in}_i(\mathit{index}_i)} < \mathit{outpos}_{\mathit{in}_i(\mathit{index}_i)}.
\end{aligned}
$$

$S_i^{\mathrm{SNAP}} :\Leftrightarrow state_i = \mathrm{SNAP} \Rightarrow$
$mode_i \neq \mathrm{TERMINATED},\ 0 \leq index_i < in_i().\mathrm{getSize}(),$
$message_i$ instanceOf SnapMessage,
$message_i = In_i(index_i)[inpos_{in_i[index_i]} - 1],$
$snap_i$.

$S_i^{\mathrm{SNAP2}} :\Leftrightarrow state_i = \mathrm{SNAP2} \Rightarrow$
$mode_i \in \{\mathrm{SNAPPING},\ \mathrm{BROADCASTING}\},$
$0 \leq index_i < in_i().\mathrm{getSize}(),$
$message_i$ instanceOf *snapMessage*,
$message_i = In_i(index_i)[inpos_{in_i[index_i]} - 1].$

$S_i^{\mathrm{SNAP3}} :\Leftrightarrow state_i = \mathrm{SNAP3} \Rightarrow$
$mode_i = \mathrm{BROADCASTING},$
$missingSnaps = 0.$

$S_i^{\mathrm{RESULT}} :\Leftrightarrow state_i = \mathrm{RESULT} \Rightarrow$
$mode_i \neq \mathrm{TERMINATED},\ 0 \leq index_i < in_i().\mathrm{getSize}(),$
$message_i$ instanceOf ResultMessage,
$message_i = In_i(index_i)[inpos_{in_i[index_i]} - 1].$

Proof of Invariance: The invariance of I can be proved by showing that the generalized proposition I' defined as the conjunction of I and of all the properties above remains invariant under each state transition. The proof of this proceeds by case destinction of of $state_i$ as illustrated in Subsection A. These proofs are rather straight-forward from the code and from I; we omit the details.

Termination

For proving termination, it is necessary to introduce the following assumptions:

1. The network is fully connected.
2. Node 0 eventually generates a snap message.
3. The operation `s.select()` is fair, i.e., it eventually returns the index of every non-empty channel in s.

The first assumption is true by construction. The second assumption is only probabilistically true (because the random generator might never trigger the generation of the snap message). The third assumption is *not* true in DAJ. If the first channel of s not empty, other channels in s are not considered. In order to guarantee termination, one thus has to implement fairness using e.g. a counter f initialized as 0:

```
InChannelSet s;
s.addChannel(in(f));
f++; if (f == in().getSize()) f = 0;
index = s.select(random(5))
```

Let us assume that all requirements are actually met. Then node 0 eventually triggers a snap message which, by connectivity and fairness, eventually reaches every other node and is forwarded to every output channel and thus to every channel of the network. Thus in every node eventually *missingSnaps* drops to 0 and a result message is generated which, again by connectivity and fairness, eventually reaches every other node. Thus in every node *missingResults* eventually drops to 0 and mode `TERMINATED` is reached, which lets the node terminate.

Time Complexity

Another question is the time complexity of the algorithm, i.e., *when* will have all nodes determined the network value (and correspondingly terminate)? The time complexity of the algorithm is the time from when node 0 triggers the snapshot until the time that the last node has determined the total result. This time is the sum of three components:

1. The time from the triggering of the snapshot until the node is notified about the snapshot.
2. The time from the node's notification of the snapshot until it has determined its *snapValue*.
3. The time from the determination of the node's *snapValue* until it has received all other node's *snapValue*s to yield *totalResult*.

One should note that different nodes may be in different phases, i.e., one node may have not yet been notified about the snapshot while another is already broadcasting its snap value.

The algorithm's time complexity can be estimated by considering the maximum number of messages that an individual node has to process in a network of size n: the node has to receive at most $i + i(n-1) + 1$ input messages (where i is the input degree of the node) and to send send $o + on$ output messages (where o is the output degree of the node). Assuming $i = o = d$, the total number of messages processed is bounded by

$$2nd + d + 1$$

For the hexagon network with 19 nodes implemented by the program, above bound gives

$$235 = 2 * 19 * 6 + 6 + 1$$

Actually the program terminates about 330 time units after node 0 has triggered the snapshot, which includes the delays until all nodes have been notified about the snapshot, the delays until all snapshot values have been propagaged through the network and delays occuring because nodes process the value messages of the normal operation and/or are blocked waiting for messages.