INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Distributed Reactive Machines

Frédéric Boussinot — Jean-Ferdy Susini — Laurent Hazard

N° 3376

Mars 1998

THÈME 1

Rapport de recherche

ISSN 0249-6399

Distributed Reactive Machines

Frédéric Boussinot , Jean-Ferdy Susini* , Laurent Hazard†

Thème 1 — Réseaux et systèmes
Projets Meije

Rapport de recherche n° 3376 — Mars 1998 — 31 pages

Abstract: One considers systems made of synchronizers to which distributed reactive machines are connected. The corresponding model is described with its implementation in Java, using SugarCubes and the RMI mechanism.

Key-words: Reactive Systems, Distribution, SugarCubes, Java, RMI

With support from France Telecom-CNET

* EMP-CMA/INRIA Meije
† CNET DTR/ASR

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, B.P. 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : 04 92 38 77 77 - International : +33 4 92 38 77 77 — Fax : 04 92 38 77 65 - International : +33 4 92 38 77 65

1. Introduction

Reactive Systems[HP] combine two main characteristics:

- They are continuously running systems, not intended to terminate. Thus, they do not fall into the class of traditional programs which are executed with some data, and which terminate after a while by producing a result. On the contrary, reactive systems interact continuously with the environment.

- In response to an activation, a reactive system reacts, depending on the environment state, by changing it, then it waits for the next activation, an so on without ever ending. Reactions to activations are called *instants*.

At the communication level, broadcast of information between parallel components has several advantages, compared to traditional communication mechanisms as message passing or *rendezvous*:

- It is simple, intuitive, and powerful; the same information is, for example, transmitted to several receivers in one single operation.

- It allows a modular approach, as new receivers can be dynamically added to the system without inducing any change to emitters.

By taking instants into account, one can define a variant of broadcast, called *instantaneous broadcast,* based on *events* defined as follows[1]:

- An event is present, absent, or undefined. It is undefined at the beginning of each new instant (events are not persistent data).

- An event cannot be both present and absent during the same instant, and once defined it remains as it during the whole instant (coherency property).

In the instantaneous broadcast paradigm, an event is received by all receivers at the very instant it is generated. Instantaneous broadcast has two more advantages over simple broadcast:

- It gives an implicit way to associate dates to event and provides an automatic synchronization on ends of instants. As a consequence, these notions do not have to be implemented if needed, as they are already present in the formalism.

- Simultaneity and absence of events are seen in a coherent way in the whole system.

In this text, one extends reactive systems to networks by defining distributed reactive systems, that one simply calls *synchronized systems*. Synchronized systems are made of components, called reactive machines, distributed over the network which share the same instants and communicate by instantaneous broadcast using a special component called *synchronizer*.

[1] The notion of an event comes from the synchronous language Esterel [BG], where it is called *signal*.

Also described in the text is the implementation of synchronized systems in the Java language[GJS], using a set of classes called SugarCubes (described below). The implementation runs on two execution *Distributed Processing Environments* (DPE): JavaRMI, and an experimental distributed platform especially designed for telecommunications.

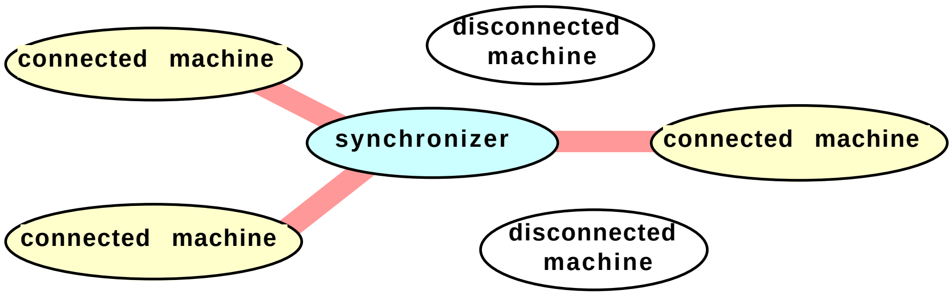
The structure of the text is as follows: in part 2 one describes the model of synchronized systems. The SugarCubes used to implement synchronized systems are briefly described in part 3. Synchronizers are defined in part 4. In part 5, one defines three new instructions for connections, disconnections, and event broadcast. Synchronized reactive machines are described in part 6. In part 7, one describes how to execute synchronized systems. Another algorithm, using counters, and its implementation on a platform different from RMI are described in part 8. Finally, one considers in part 9 several related works et one makes proposals for future work.

2. Synchronized Systems

One starts by defining the model of synchronized systems, then the algorithm to detect ends of instants used to implement them.

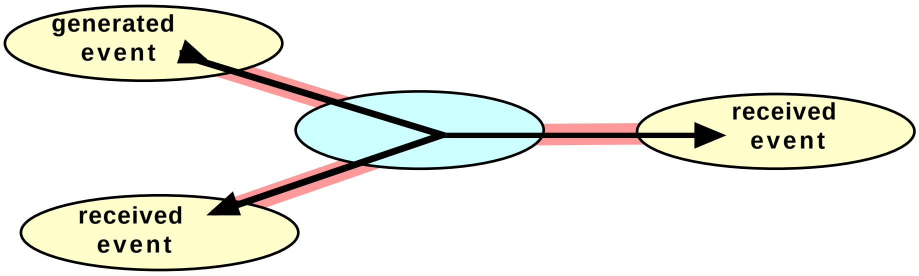
The Model

One considers systems made of a synchronizer to which are connected reactive machines distributed over the network. Reactive machines connected to a synchronizer all execute at the same pace and share the same instants; on the contrary, machines which are not connected to any synchronizer execute at their own pace. A machine cannot be connected to several synchronizers at the same time.



A synchronizer with three connected machines

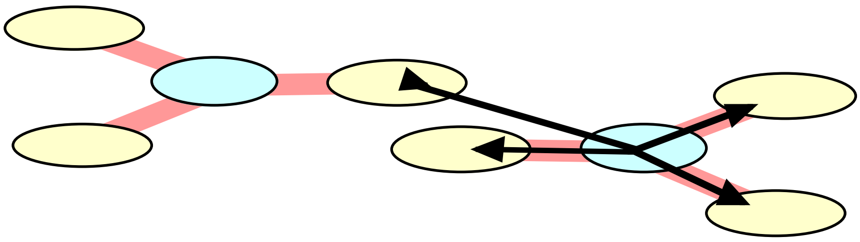
Any machine has the possibility to broadcast an event to the machines connected to a synchronizer. This broadcast is coherent; all machines connected to the synchronizer receive the event at the same instant; moreover, if the emitting machine is also connected to the synchronizer, the instant of emission is the same as the instant of reception: in this case, broadcast is instantaneous.



Synchronous broadcast of an event

Actually, systems made of machines linked to synchronizers are some kind of *dynamic reactive areas* in which communication is broadcast. Moreover, broadcast is instantaneous inside the same area (when the emitting and the receiving machines are connected to the same synchronizer).

Note that communication is asynchronous between distinct areas: the instant of reception is not necessarily the one of emission.



Asynchronous broadcast of an event, between distinct areas

Reactive machines can dynamically connect to a synchronizer and disconnect from it during execution. There are thus two ways for a machine to communicate with a remote area: either by directly sending an event to the remote synchronizer (asynchronous broadcast), or by connecting to it before sending the event (synchronous broadcast).

Determining ends of instants

Implementation is based on an algorithm decomposed in phases to determine ends of instants.

At the beginning of each phase, the synchronizer waits for all connected machines to end execution for the current instant, or to suspend execution, awaiting some events. During this, it stores events that are to be broadcast. When all machines have terminated or are suspended, the synchronizer ends the phase by sending all events to be broadcast to each suspended machine[2]. At the end of a phase, when there is no event to

[2] It is useless to send them to machines which have already terminated.

broadcast, the synchronizer decides that the current instant is over and sends a signal to all suspended machines to indicate that the next instant can start.

Disconnection of a machine from the synchronizer to which it is connected is postponed to the end of the global instant. This guaranties that the machine always has a coherent vision of events broadcast by the synchronizer and that there is no risk that it does not receive an event because disconnection is too soon.

Connection of a free machine to a synchronizer is postponed to the beginning of the next machine instant to let the machine and the synchronizer synchronize on the same instant.

3. SugarCubes

The aim of the *reactive approach* is to propose a flexible programming of reactive systems, especially those which are dynamic (that is, the number of components and their connections are changing during execution). Informations on this approach can be found on the Web at the URL `http://www.inria.fr/meije/rc/`.

The Reactive-C[] language was the first formalism developed following this approach. Reactive-C is an extension of the C programming language to program reactive systems. The main Reactive-C primitives have recently been ported to Java as a set of classes named *SugarCubes*[BS].

The two main notions of SugarCubes are the one of *reactive instruction* whose semantics refer to instants, and the one of *reactive machine* whose purpose is to execute reactive instructions in an environment made of instantaneously broadcast events.

The Class Instruction

The `Instruction` class implements reactive instructions. A reactive instruction can be activated (method `activ`), reset (method `reset`), or forced to terminate (method `terminate`). Each activation returns as result one of the three following values:

- TERM (for *terminated*) means that the instruction is completely terminated; nothing remains to do for the current instant and also for future ones. Thus, to activate an other time an instruction returning TERM has no effect and returns also TERM.

- STOP (for *stopped*) means that execution of the instruction is over for current instant, but that code remains to be executed at next instants.

- SUSP (for *suspended*) means that execution of the instruction has not reached a stable state and must be resumed during current instant. This is for example the case for the instruction that waits for a not yet generated event (see below): execution is suspended to let the other components the possibility to generate the event during current instant.

A call to method `terminate` forces the instruction to completely terminate and thus to return TERM when activated.

A call to method `reset` resets the instruction which thus returns in its initial state.

The basic reactive instruction of SugarCubes are:

- `Stop`, which stops execution for the current instant;
- `Seq` to put one reactive instruction in sequence with another one;
- `Merge` to put two reactive instructions in parallel;
- *atoms* to execute basic Java statements such as printing messages;
- `Loop` and `Repeat`, for cyclic executions;
- `Generate` to generate an event, and `Await` to wait for it.

The Class Machine

The class `Machine` implements reactive machines. A reactive machine executes a program which is a reactive instruction. It has two main tasks to perform: first, to decide the ends of instants, and second, to deal with broadcast events. Initially, the program is the `Nothing` instruction which does nothing and terminates instantaneously. New instructions are dynamically added to the program (by calling the machine method `add`) and executed in parallel with the previous ones.

Basically, a reactive machine detects the end of the current instant, that is when all parallel instructions of the program are terminated or stopped. The behavior is as follows:

- The program is cyclically activated while there are suspended instructions in it (that is, while its activation returns SUSP).
- The end of the current instant is effective when all the parallel instructions in the program are terminated or stopped (no suspended instruction remains).
- At the end of each program activation, the machine tests if some new events were generated during this execution. If it was not the case, then there is no hope that future program activations will change the situation, and the end of the current instant can be safely decided. Then, a flag is set to let suspended instructions stop, knowing from that point that awaited events are absent.

Two variables `move` and `endOfInstant` are used to implement this behavior. The variable `move` is set to true to indicate that a new event is generated (`Generate` statement); in this case, the end of the current instant is postponed to allow the suspended receivers awaiting the event (`Await` instruction) to resume. The `endOfInstant` variable is set to true when the end of the current instant is decided by the machine, to let suspended receivers know that awaited event are absent.

The method `activation` of the class `Machine` implements this behavior; the code is the following:

```
protected byte activation(Machine machine){   program activation
  byte res;                                    in the machine (this)
  endOfInstant = move = false;                 while suspended
  while(SUSP == (res = program.activ(this))){
    if(move) {move = false; continue;}
    endOfInstant = true;                       continue, if move is true
  }
  setNewProgram();                             otherwise, decide the end of instant
  newInstant();                                new instant set
  return res;
}
```

In the following, the problem will be to adapt the processing of `move` and `endOfInstant` to the case of distributed machines. Indeed, the move variable of a machine will have the possibility to be set from outside it, when new events are broadcast by other machines. Also, all endOfInstant variables are to be set in a synchronized way in order to implement the sharing of instants.

4. The synchronizer

One first define the interface Synchronizer, then the class `Synchronizer_Impl` of synchronizers.

The Synchronizer Interface

The interface `Synchronizer` extends interface `Remote` (see [RMI]) and defines the following methods:

- `connect` connects the machine which is given as parameter. It returns a number that identifies the machine.
- `disconnect` disconnects the machine whose number is given as parameter.
- `broadcast` broadcasts the event whose name is given as parameter to all connected machines.
- `suspended` signals that the machine whose number is given as parameter is suspended.
- `completed` signals that the machine whose number is given as parameter has terminated its execution for the current instant.

All these methods declare the exception `RemoteException` of JavaRMI.

The code of the interface `Synchronizer` is the following:

remote interface

```
public interface Synchronizer extends Remote
{
  public int  connect(MachineSync mach)  throws RemoteException;
  public void disconnect(int num)        throws RemoteException;
  public void broadcast(String event)    throws RemoteException;
  public void suspended(int num)         throws RemoteException;
  public void completed(int num)         throws RemoteException;
}
```

MachineSync is the type of synchronized machines

all methods declare RemoteException

Synchronizer Implementation

Class `Synchronizer_Impl` implements the two interfaces `Synchronizer` and `Runnable`. It starts by the definition of the maximum number of machines that can be connected simultaneously to the synchronizer (presently 5), and by the definition of four constants to code machine states:

```
final static int MaxMachineNumber = 5;
final static byte undef        = 0;
final static byte suspend      = 1;
final static byte completed    = 2;
final static byte disconnected = 3;
```

undefined status

suspended machine

execution of the machine is terminated for the current instant

disconnected status

Note that the maximum number of simultaneously connected machines can be changed without any problem.

The following fields are defined to deal with connected machines:

number of machines actually connected

```
protected int numberOfMachines = 0;
protected MachineSync[] machines = new MachineSync[MaxMachineNumber];
protected byte status[] = new byte[MaxMachineNumber];
```

machine array

machine state array

The vector `broadcastDemand` stores the broadcast demands received during the current phase; the vector `broadcastSum` stores all broadcast demands received up to now since the beginning of the current instant:

vector of not yet done broadcast demands

```
protected Vector broadcastDemands = new Vector();
protected Vector broadcastSum = new Vector();
```

vector of all broadcast demands

Constructor

The unique constructor of `Synchronizer_Impl` assigns the value `disconnected` to all elements of the array `status`:

```
public Synchronizer_Impl(){
  for(int i = 0; i < MaxMachineNumber; i++){
     status[i] = disconnected;
  }
}
```

Connections and Disconnections

Disconnecting a machine simply means to put its status to `disconnected`:

```
public synchronized void disconnect(int num){
  if (status[num] == disconnected) return;
  status[num] = disconnected;
  numberOfMachines--;
}
```

Method `disconnect` must be synchronized because it changes the vector `status` which is also used by the synchronizer, during its execution.

During connection, if the maximum number of simultaneously connected machines is not exceeded, one takes the first free slots in the two arrays `machines` and `status` and place in them the machine with the `undef` status. One also broadcast to the machine all events contained in `broadcastSum`, broadcast during previous phases; thus, the new connected machine has the same vision of broadcast events than the other connected ones. Method `connect` must be synchronized for the same reasons `disconnect` is.

```
public synchronized int connect(MachineSync machine){
   if (numberOfMachines != MaxMachineNumber){
     for(int i = 0; i < MaxMachineNumber; i++){
       if (status[i] == disconnected){
         machines[i] = machine;
         status[i] = undef;
         numberOfMachines++;
         if (numberOfMachines != 1 && broadcastSum.size()>0){
            try{ machines[i].generate(broadcastSum); }
            catch(RemoteException e){System.out.println(e);}
         }else{
            broadcastDemands.removeAllElements();
            broadcastSum.removeAllElements();
            notifyAll();
         }
         return i;
       }
     }
   }
  return DISCONNECTED;
}
```

connection of the machine

broadcast demands, during the instant

it's the only connected machine: reset of previous broadcast demands

notification for restarting the synchronizer

too many connected machines

Broadcast of Events

The method `broadcast` put the event to be broadcast into the vector `broadcastDemands`, if it is the first time during the instant this demand is made (it is not an element of `broadcastSum`). Actual event broadcast will be done later, at the end of the phase, by the method `broadcastProcessing`.

Code for method `broadcast` is the following:

```
public void broadcast(String event) {
   if (broadcastSum.contains(event)) return;
   broadcastSum.addElement(event);
   broadcastDemands.addElement(event);
}
```

Method `broadcastProcessing` broadcast the same event vector (`toSend`) to all suspended machines, and also gives them the status `undef`. Note that a copy of `broadcastDemands` must be performed as machines can immediately make new demands, and thus transform `broadcastDemands`, before the end of `broadcastProcessing`.

Code for `broadcastProcessing` is the following:

copy of broadcast demands

```
public void broadcastProcessing() {
  Vector toSend = (Vector)broadcastDemands.clone();
  broadcastDemands.removeAllElements();
  for(int i = 0; i < MaxMachineNumber; i++){
    if (status[i] != disconnected && status[i] != completed){
      status[i] = undef;
      try{machines[i].generate(toSend);}
      catch(RemoteException e){System.out.println(e);}
    }
  }
}
```

machine status set undefined

broadcast of demands

Method `broadcastProcessing` does not need to be synchronized as it is only called by `step` (see below) which is.

Suspension and Completion

Methods `completed` and `suspended` associate the corresponding status to the machine given as parameter. Here is the code for `completed` (the one for `suspended` is very similar):

```
public synchronized void completed(int num){
  status[num] = completed;
}
```

Methods `connect` and `disconnect` are both synchronized as they transform `status` which is also used by the synchronizer.

Execution Step

Method `step` is the central one which links up phases and instants. Event broadcast is made after reception of the status from all connected machines. The global transition to the next instant is performed when there is no more event to broadcast. Code for `step` is the following:

```
protected synchronized void step(){
  for(int i = 0; i < MaxMachineNumber; i++){
    if (status[i] != disconnected && status[i] == undef) return;
  }  <---- all machines have sent their status
  if (broadcastDemands.size() > 0){
    broadcastProcessing();
    return;
  }  <---- there is nothing to broadcast
  broadcastSum.removeAllElements():
  instantIsOver();  <---- end of the global instant
}
```

Method `instantIsOver` signals the end of the current global instant to each of the connected machines and sets their status to `undef`:

```
protected void instantIsOver(){        signals the end of instant
  for(int i = 0; i < MaxMachineNumber; i++){
    if (status[i] != disconnected){
      try { machines[i].instantIsOver(); }  <----
      catch(RemoteException e){ System.out.println(e); }
      status[i] = undef;
    }     ^---- reset of the machine status
  }
}
```

Note that `instantIsOver` is only called by `step` which is synchronized, and thus it does not need to be so.

Running

The method `run` of the class `Synchronizer` is an infinite loop which suspends execution while no machine is connected (method `waitAtLeastOne`), and which otherwise executes `step`:

```
                 to let other threads the possibility to execute
public void run (){                                      |
  while(true){ waitAtLeastOne(); step(); Thread.yield(); }
}
```

The method `waitAtLeastOne` suspends the synchronizer execution while no machine is connected. Execution resumes when a notification is sent by method `connect` (execution of `notifyAll`).

Code for `waitAtLeastOne` is the following:

```
protected synchronized void waitAlLeastOne (){
  if (numberOfMachines == 0){
     while (numberOfMachines == 0){
        try{wait();}  ← waiting terminated by a connection
        catch(InterruptedException e){System.out.println(e);}
     }
  }
}
```

Finally, the method `launch` launches a thread to execute `run` (recall that `Synchronizer` implements `Runnable`):

```
public void launch(){ new Thread(this).start(); }
```

5. New Instructions

One defines three new reactive instructions: `Connect` which asks for a connection to the synchronizer associated to an URL, `Disconnect` which disconnects the machine, and `Broadcast` which generates an event and broadcasts it to all the machines connected to a synchronizer associated to an URL.

Instruction `Disconnect` asks for the disconnection and terminates at next instant.

Code for `Disconnect` is the following:

```
public class Disconnect extends Instruction
{
  protected byte activation(Machine machine){
    ((MachineSyncImplem)machine).askDisconnection();
    terminate();
    return STOP;   asks for disconnection
  }
}        terminates at next instant
```

Instruction `Connect` asks for the connection to the synchronizer whose URL is given as parameter; it stops while the connection is not performed:

```
public class Connect extends Instruction
{
  protected String url;
  public Connect(String url){ this.url = url; }
  protected byte activation(Machine mach){
    MachineSyncImplem machine = ((MachineSyncImplem)mach;
    if (machine.connected()) terminate();    <- termination when the
    else machine.askConnection(url);            connection is done
    return STOP;   <- stops in all cases
  }
}
```

The instruction `Broadcast` extends `Generate` and calls the method `broadcast` of the synchronizer whose URL is given as parameter at construction:

```
public class Broadcast extends Generate
{
  protected Synchronizer sync;
  protected String url;   <- URL of the synchronizer
  public Broadcast(String event,String url){
    super(event); this.url = url;
  }                   <- finding the synchronizer
  protected boolean findSynchronizer(){
    try{sync = (Synchronizer)Naming.lookup(url); return true;}
    catch(Exception e){System.out.println(“”); return false;}
  }
  protected void action(Machine machine){
    super.action(machine);
    if (sync == null && !findSynchronizer()) return;
    try { sync.broadcast(eventName); }
    catch(RemoteException e) { System.out.println(e); }
  }      call to the synchronizer to broadcast the event
}
```

We choose to determine the synchronizer at run time, when the instruction is executed. An other solution would be to associate the synchronizer to the machine. This solution would be more efficient (the synchronizer needs not to be computed at each execution of the instruction) but less flexible (all instructions `Broadcast` would share the same synchronizer).

6. Synchronized Machines

First, one defines the remote interface `MachineSync` then the class `MachineSyncImplem` of synchronized reactive machines.

Interface MachineSync

Interface `MachineSync` extends `Remote` and defines the two following methods:

- `instantIsOver` signals that instant is over to the machine.

- `generate` signals that the events elements of the vector given as parameter are broadcast.

Code for `MachineSync` is:

```
public interface MachineSync extends Remote
{
  public void instantIsOver()              throws RemoteException;
  public void generate(Vector eventList) throws RemoteException;
}
```

Implementation of Machines

The class `MachineSyncImplem` extends `Machine` and implements the two interfaces `MachineSync` and `Runnable`. The following fields are defined:

```
protected String name;          <- machine name
protected int num = DISCONNECTED;   <- machine number
protected Synchronizer synchronizer = null;   the associated
protected String url;          <- synchronizer and
protected boolean connectionAsked = false,    its URL
                  disconnectionAsked = false;
```

The method `instantIsOver` sets to true the field `endOfInstant` of the machine which means that the current instant is over. The method `generate` generates the events which are elements of the vector given as parameter, then sets `move` to true to indicate that new events are present:

```
public void instantIsOver(){
  synchronized (this){ endOfInstant = true; notifyAll(); }
}                                          list of events
public void generate(Vector eventList){    to broadcast
  while(eventList.size() > 0){
    generate((String)eventList.firstElement());
    eventList.removeElementAt(0);
  }                       event generation
  synchronized (this){ move = true; notifyAll();};
}
                       there is a change in the machine
```

Note that the two variables `endOfInstant` and `move` have to be protected because they are also used by the machine.

Connections and Disconnections

The two methods for connection and disconnection are the following:

- `connect` tries to connect the machine to the synchronizer which is in parameter. The result indicates if the connection is performed or not.

- `disconnect` disconnects the machine from the associated synchronizer.

Here is the code for `askConnection` and `connect` (the one for the disconnection methods is very similar and is not given):

```
public void askConnection(String url){
  if (connected()) return;
  this.url = url; connectionAsked = true;
}
                                 finding the synchronizer
protected void connect(){
  try {
    synchronizer = (Synchronizer)Naming.lookup(url);
    int res = synchronizer.connect(this);  <-- connection attempt
    if (res != DISCONNECTED) num = res;  <-- connection done
  }catch(Exception e){}
}
```

Activations

The method `activation` uses the field `move` to test if a new event is present, coming either from the machine, or from the synchronizer by a call to `generate`. This implies that `move` must be protected.

Code for method `activation` is the following:

```
protected byte activation(Machine machine){
  byte res;  possible connection at the beginning of the instant
  endOfInstant = move = false;
  if (connectionAsked){connect(); connectionAsked = false;}
  while(SUSP == (res = program.activ(this))){
    synchronized (this){if(move) {move = false; continue;}}
    waitOverOrMove();        <-- waiting for the end of the
  }                              instant or for new events
  if (!endOfInstant) waitOver();
  setNewProgram();         <-- waiting for the end of instant
  newInstant();
  if (disconnectionAsked){disconnect(); disconnectionAsked = false;}
  return res;  <-- possible disconnection at the end of the instant
}
```

Method `waitOverOrMove` signals machine suspension to the synchronizer and waits in return for a signal from it to indicate the end of the current instant or the generation of a new event:

end of instant if no synchronizer is present

```
protected void waitOverOrMove(){
  if (!connected()){ endOfInstant = true; return; }
  try { synchronizer.suspended(num); }
  catch(RemoteException e){ System.out.println(e); }
  synchronized(this){
    while (!endOfInstant && !move){
      try{wait();}
      catch(InterruptedException(e){System.out.println(e);}
    }
  }
}
```

signals suspension to the synchronizer

waiting for the end of instant or a new event

Method `waitOver` is similar except that it signals the termination of the machine by calling the method `completed`, instead of `suspended`, and only waits for the end of instant.

Running

The method `run` starts by calling the protected method `init` whose purpose is to initialize the machine program, then it cyclically activates it.

After setting the machine name, method `launch` starts a thread to execute `run`:

to let other threads execute

```
public void run (){
   init(); while (activ(this) != TERM) Thread.yield();
}

public void launch(String name){
  this.name = name; new Thread(this).start();
}
```

the program is put into the machine

starts a thread to execute run

Here is the example of a definition of `init` which adds to the machine an instruction that runs five connection/disconnection steps, and that prints a message at each instant (instruction `PrintStep` is not described):

```
protected void init(){
  add(new Repeat(5,          adds the instruction in the machine
     new Seq(
        new Seq(             URL of the synchronizer
           new Connect(url),
           new Repeat(100,new Seq(new PrintStep(),new Stop()))),
        new Seq(
           new Disconnect(),
           new Repeat(100,new Seq(new PrintStep(),new Stop()))))));
}
```

7. System Execution

Using the previous classes and the Java RMI mechanism, one defines two new executable classes: `Sync` which implements synchronizer objects and `Mach` which implements synchronized reactive machines. Both classes implements `UnicastRemoteObject`.

The Class Sync

When run, an object of type `Sync` receives as parameter the name of the physical machine on which the synchronizer is launched and its name. These two informations form an URL of the form `//syncMachine/syncName` which identifies the synchronizer.

The method `main` of class `Sync` performs the following actions:

- Execution of a *security manager*.
- Creation of a synchronizer of type `Synchroniser_Impl`.
- Execution of the synchronizer (method `launch`).
- Export of the synchronizer (`Synchroniser_Impl` does not extends `UnicastRemoteServer`; thus, the export must be explicit).
- Record of the synchronizer in the name server of RMI.

Code is as follows:

```
                                     installation of a new security manager
System.setSecurityManager(new RMISecurityManager());
try {                                          new synchronizer
   Synchronizer_Impl sync = new Synchronizer_Impl();
   sync.launch();      launch of the synchronizer
   exportObject(sync);   export of the objet of type Synchronizer_Impl
   String url = "//"+syncMachine+"/"+syncName;
   Naming.rebind(url,sync);                       records the
}catch(Exception e){ System.out.println(e); }    synchronizer in the
                                                  name server
```

The class Mach

When executed, an object of the class `Mach` receives as parameter its name, the one of the physical machine on which a synchronizer runs, and the name of it.

The method `main` of the class `Mach` performs the following actions:

- Creation of a synchronized machine.
- Export of this machine.
- Transmission to the machine of the synchronizer URL (method `linkTo` of the class `MachineSync_Impl`).
- Launch of the machine (method `launch`).

The code is:

creation of a machine which extends MachineSyncImplem

```
try {
   String url = "//"+syncMachine+"/"+syncName;
   MachineSync machine = new MachineSync_Impl();
   exportObject(machine);
   machine.linkTo(url);           <-- machine linked to the synchronizer
   machine.launch(machineName);   <-- the machine is launched
}catch(Exception e){ System.out.println(e); }
```

Execution

Now, one describes the list of commands needed to run the executable files. First, one launches the JavaRMI name server by the command `rmiregistry`. Second, the synchronizer is run by a command of the form:

physical machine on which the synchronizer runs

```
java Sync urna.inria.fr synchroniseur
```

synchronizer name

Finally, synchronized machines are launched by commands of the form:

synchronized machine name
synchronizer name

```
java Mach M1 urna.inria.fr synchroniseur
```

physical machine on which the synchronizer runs

Note that the synchronized machines and the synchronizer can be run on distinct physical machines.

8. Another Algorithm, another DPE

In this section, we present a different algorithm for implementing distributed instants which can be implemented on distributed execution platforms others than RMI.

An Algorithm based on Counters

The basics of the protocol used by the algorithm are the followings:

- Each machine manages a counter of received messages from the synchronizer. A message is a method invocation which signals the end of the current instant: method `instantIsOver`, or a broadcast event: method `generateBroadcast` (note that one uses a new method distinct from `generate`). The counter is reset when the machine connects to the synchronizer.

- At the end of each phase, the machine sends a message to the synchronizer. This message invokes one of the methods `suspended` or `completed`; as a parameter, it contains the number of received messages from the synchronizer which have been already processed during the phase.

- During execution of a phase, a machine can ask the synchronizer to broadcast one event to other connected machines.

- For each connected machine, the synchronizer manages a counter of all messages sent to it.

- The synchronizer proceeds broadcast event requests in the following way: the event is sent to all connected machines (except the one that sends the event) having (i) signalled the end of a phase, which means that the machine has processed all messages sent to it, and (ii) is suspended (and thus can potentially use the event).

- The synchronizer sends the message indicating end of instant when (i) all connected machine have signalled the end of a phase, and (ii) there is no more broadcast event requests.

Several points must be noticed:

1. This protocol does not need messages to be processed in a synchronous way as *interrogations*, and allows one to implement them as *oneway* invocations.

2. However, a condition must be satisfied by the distributed processing environment: messages must be processed by a server in the same order they are sent by a client. This is crucial for the synchronizer to avoid deciding the end of the current instant before having processed all broadcast demands. This order is automatically preserved by interrogations (this is the case with RMI). If the platform does not guaranty this, the method `broadcast` cannot be declared as oneway.

3. As with the previous algorithm, this protocol does not support message loss. But by contrast with the previous one that can be called "*silent fail*" (a message loss blocks

execution), the new protocol can produce false results, if method `bradcast` is one-way and a message for it is lost.

Algorithm Implementation

Some changes are needed in the previous implementation to deal with the new protocol. We present the ones concerning the interface `MachineSync` and the class `MachineSync-Implem`.

Interface MachineSync

Interface `MachineSync` becomes:

```
public interface MachineSync extends Remote {
  public void instantIsOver()             throws RemoteException;
  public void generateBroadcast(String event) throws RemoteException;
}
```

method called by the synchronizer

Class MachineSyncImplem

The class `MachineSyncImplem` is changed as follows:

counter of awaited messages

```
  protected int shouldReactTo = 0;
  protected int extMoves = 0;
```

counter of received messages

```
  public synchronized void instantIsOver () {
    instantOver = true; notifyAll ();
  }
  public void generateBroadcast (String event){
    super.generate(event); newMoveB();
  }
```

synchronized methods to use move

```
  ...
  public synchronized void newMove() { move = true; notify ();}
  public synchronized void newMoveB() {
     extMoves += 1; move = true; notify ();
  }
  public synchronized int resetMove() {
    move = false;
    shouldReactTo += extMoves; extMoves = 0;
    return shouldReactTo;
  }
  public synchronized boolean hasMove() { return move;}
```

The methods waiting for signals from the synchronizer are changed to process the counter value (one only give code for `waitOverOrMove`):

```
protected void waitOverOrMove(int no){
  if (connected()) {
    synchronized (this) {
      try { synchronizer.suspended(num, no); }
      catch (RemoteException e) { num = DISCONNECTED;}
      while (!instantOver && !hasMove()){
         try{wait();}
         catch(InterruptedException e){System.out.println(e);}
      }
    }
    if (instantOver) endOfInstant = true;
  }else endOfInstant = true;
}
```

number of awaited messages

The method to activate the machine is transformed in the same way to deal with counters and also to systematically send the message `completed` (in order to update the synchronizer counter):

```
protected byte activation(Machine machine){
  byte res = TERM;
  boolean encore = true;
  int reactingTo = 0;
  endOfInstant = false;
  if (connectionAsked) { connect(); connectionAsked = false; }
  while (encore) {
    reactingTo = resetMove();
    res = program.activ(this);
    if (res == SUSP) {
      if (!hasMove ()) waitOverOrMove(reactingTo);
    }  else encore = false;
  }
  waitOverAndReset (reactingTo);
  if (disconnectionAsked){ disconnect(); disconnectionAsked = false; }
  setNewProgram();
  return res;
}
```

value of the counter of awaited messages

systematic sending of completed

Interface Synchronizer

The interface `Synchronizer` is simply changed to introduce counters:

```
public interface Synchronizer extends Remote
{
  ...
  public void suspended(int num, int no) throws RemoteException;
  public void completed(int num, int no) throws RemoteException;
}
```

Class Synchronizer_Impl

Introduction of counters induces changes in the code of class `Synchronizer_Impl`. For each machine, two variables are defined: `shouldReactTo` which counts messages sent to the machine and `reactedTo` which stores the last value sent by the machine.

A method `waitReacted` is defined: if there exists at least one connected machine, it waits that all connected machines have reacted to all sent messages; it returns a boolean (true, if there exists a connected machine, false otherwise):

```
protected synchronized boolean waitReacted () {
  boolean mustWait;
  int cur = -1;                         the machine has not finish to process messages sent
  while (true) {
    mustWait = false;
    if (numberOfMachines > 0) {
      for(int i = 0; i < MaxMachineNumber; i++){
        if ((status[i] != disconnected) &&
            (reactedTo[i] < shouldReactTo[i])){
              mustWait = true; cur = i; break;
        }
      }                         no connected machine
    }else return false;
    if (mustWait){
      try{wait();}
      catch(InterruptedException e) {System.out.println(e);}
    }else break;
  }
  return true;        there exists connected machines
}
```

This method is used in the main loop of the thread in `Synchronizer_Impl` (broadcast demands are put in the vector posted, and are processed by `sendGenerate`):

```
public void run (){
  boolean hasWork;
  while(true){
    waitAtLeastOne ();   event broadcast
    hasWork = true;                waiting a situation where all
    while (hasWork) {              machines have react to all
      sendGenerate();              messages
      hasWork = waitReacted();
      if (hasWork) hasWork = !posted.isEmpty ();
      else posted.removeAllElements ();
    }                     send end of instant to all machines
    instantIsOver();
  }
}
```

Others methods of the class -connection, disconnection processing and event broadcast- are not presented here.

A different Platform

The RMI mechanism has many advantages: it is simple to use, freely distributed with the JDK1.1 and relatively efficient as close to Java. However, it has some disadvantages: it does not allow cooperation with other DPE (at least in the present version) and implements only synchronous communication (interrogations). As a consequence, the client of an interface remains blocked while the invocation has not been processed by the server, even if no result is needed. This is an obstacle to true concurrent executions of distributed applications as the one of synchronized reactive machines where there are many interactions.

The presence of asynchronous method calls (notifications) in DPE increases parallelism and concurrency. In CORBA compliant platforms methods can be declared oneway in interfaces exported by servers objects. We have tested our implementation of synchronized reactive machines in an experimental CORBA platform.

In CORBA, one describes server interfaces using the *Interface Description Language* (IDL). The interfaces `MachineSync` and `Synchronizer` are described as follows:

```
module ReactiveMachines {
   interface MachineSync {
        oneway void instantIsOver();
        oneway void generateBroadcast (in string ev);
   };

   interface Synchronizer {
        long connect(in MachineSync mach);
        void disconnect(in long num);
        oneway void broadcast(in string ev,in long from);
        oneway void suspended(in long num, in long no);
        oneway void completed(in long num, in long no);
   };
};
```

As previously said, method `broadcast` can only be declared as oneway if invocations are processed in the order they are sent. This is true for the considered DPE on which a unique thread is used to process invocations sent to one server. Invocations from the same client are sent and received using one single TCP/IP connection and are thus processed in the emission order by the unique thread in the server object.

The classes `MachineSyncImplem` and `Synchronizer_Impl` are modified to conform to (i) interface implementations, and (ii) to the way remote interfaces are referenced in the new platform. These minor changes are not presented here.

On the example of a simple application, one gets a gain of about 30% in the execution time when using oneway methods. However, this result is to be considered with care as the result could be different for others applications or others hardware configurations (number of processors or of physical machines).

The use of another DPE has also shown a default of RMI. Indeed, when clients and server of an interface are on the same address space (that is, the same Java virtual machine), RMI does not implement method invocation as a simple method call (which is possible and sufficient), but executes the same procedure as for external invocations (marshalling of arguments, execution of the transport protocol, *etc*...).

9. Related and Future Works

Termination Algorithms

The previously described algorithms for determining ends of instants are members of the family of *distributed termination algorithms* (see [Ma] for a presentation). They have the following characteristics:

- They are designed for structures having the form of “stars”, in which the synchronizer has the key role of starting the end of instants detection algorithm.

- They are suited for dynamic systems in which machines can connect and disconnect at every moment.

- The first algorithm is a synchronous one: JavaRMI remote method calls are synchronous calls in which the caller waits for the callee termination to resume. Synchronous algorithms are *a priori* simpler to implement than asynchronous ones; in particular, messages need no to be counted (see [Ma]).

- The second algorithm which uses counters can also work in an asynchronous context where remote methods are notified.

Synchronous Code Distribution

Synchronous code distribution has been studied for the language Lustre[HCPR] by Caspi and Girault in [CG]. Their technique can be used for Esterel *via* the .oc format code which is common to these languages. From an other point of view, the CRP formalism[BRS] is an attempt to distribute Esterel code using *rendezvous* communications.

In the system Saturn[BA], developed at Cert/Onera, broadcasting of an information to distributed synchronous modules always takes one instant. This simplifies implementation as the broadcast of an event becomes effective only at next instant.

Future Works

We plan to explore several points:

- Implementation of failure detection mechanisms (failures of connected machines must not block the system).

- Time exceed detection, during reactions of machines connected to a synchronizer. Actually, this is part of telecom research on *quality of services*[St].

- Implementation of migration facilities to allow for example a reactive instruction executed by a machine connected to a synchronizer to be transferred on an other connected machine.

10. Conclusion

We have defined distributed reactive systems made of reactive machines which dynamically connect to synchronizers and disconnect from them. Machines connected to the same synchronizer all execute at the same pace and communicate using instantaneously broadcast events (synchronous broadcast). A machine can also broadcast events to machines connected to a remote synchronizer (asynchronous broadcast).

Architectures of synchronized machines are dynamic ones (new machines and new synchronizers can be added at each moment; machines can change their connections to synchronizers at run time), but do not allow migration: reactive machines and synchronizers all stay on the physical machine on which they have been launched. However, this kind of programming, based on connections and disconnections, and on moves from one reactive area to an other, is quite close from the one of agents migrating through the network.

We have implemented distributed reactive systems with Java, using the SugarCubes, on two distinct platforms: JavaRMI on which we use a synchronous distributed termination algorithm, and an experimental distributed execution platform on which we use an asynchronous algorithm based on counters and notifications.

Bibliography

[BG] G. Berry, G. Gonthier, *The Esterel Synchronous Language: Design, Semantics, Implementation,* Science of Computer Programming, **19**(2), 1992.

[BRS] G. Berry, S. Ramesh, R.K. Shyamasundar, *Communicating Reactive Processes*, Proc. 20th ACM POPL, Charleston, Virginia, 1993.

[BA] F. Boniol, M. Adelantado, *Programming distributed reactive systems: a strong and weak synchronous coupling*, 7th International Workshop on Distributed Algorithms WDAG'93, LNCS 725, 1993.

[Bo] F. Boussinot, *Reactive-C: An extension of C to program reactive systems*, Software Practice and Experience, **21**(4): 401-428, 1991.

[BS] F. Boussinot, J-F Susini, *The SugarCubes Tool Box - Definition*, Inria Research Report 3247, available at the URL `http://www.inria.fr/meije/ rc/SugarCubes/`, 1997.

[CG] P. Caspi, A. Girault, *Distributing reactive systems*, International Conference on Parallel and Distributed Computing Systems, Las Vegas, 1994.

[GJS] J. Gosling, B. Joy, G. Steele, *The Java Language Specification*, Addison-Wesley, 1996.

[HCPR] N. Halbwachs, P. Caspi, P. Raymond, Ch. Ratel, *The Synchronous Dataflow Programming Language Lustre*, Proc. IEEE, **79**(9), 1991.

[HP] D. Harel, A. Pnueli, *On the Development of Reactive Systems*, NATO ASI Series F, Vol. 13, Springer-Verlag, 1985.

[Ma] F. Mattern, *Algorithms for distributed termination detection*, Distributed Computing, **2**:161-175, 1987.

[RMI] *Java Remote Method Invocation Specification-JDK1.1*, 1997. The site for RMI is at `http://java.sun.com/products/jdk/rmi`.

[St] J-B Stefani, *Computational aspects of QoS in an object based distributed system architecture*, 3rd International Workshop on Responsive Computer Systems, Lincoln, NH, USA, 1993.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 Villers lès Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot St Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, B.P. 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399