ID2203 Tutorial 4

Cosmin Arad
`icarad@kth.se`

Handout: February 20, 2008
Due: February 27, 2008

1 Introduction

The goal of this tutorial is to understand and get accustomed to implementing *atomic register* shared memory abstractions for the synchronous and asynchronous system models.

2 Assignment

Using the TBN framework, you will implement two atomic register abstractions as components: a fail-stop atomic register with multiple writers, described in Algorithm 1 and Algorithm 2 (Read-Impose Write-Consult), and a fail-silent atomic register with multiple writers, described in Algorithm 3 and Algorithm 4 (Read-Impose Write-Consult-Majority).

You will use the same type of input as in the previous assignments, to specify your network of processes, i.e., a topology.xml file. You will reuse the communication component, the timer component, the delay component, the beb broadcast component, and the pfd component. You have to implement a FailStopAtomicRegisterComponent (RIWC) and a FailSilentAtomicRegisterComponent (RIWCM), together with the events that are accepted and triggered by these components: AtomicRegisterReadEvent, AtomicRegisterReadReturnEvent, AtomicRegisterWriteEvent, AtomicRegisterWriteReturnEvent and the messages that they exchange: ReadMessage, WriteMessage, AckMessage, and ReadValueMessage.

You will have a separate architecture for testing the RIWC and RIWCM, i.e., an `assignment4-riwc.xml` and an `assignment4-riwcm.xml`. When launching the Assignment4.java you pass a command line argument to specify which architecture to use, and a string containing the register operations issued by the application component:

`java Assignment4 <topology.xml> <nodeId> riwc|riwcm <ops>`.

You will slightly change the application component from the third assignment, to handle the AtomicRegisterReadReturnEvent and the AtomicRegisterWriteReturnEvent, and an ApplicationEvent raised by the main program and handled by the application component, that contains the string `ops`. The `ops` string contains read and write operations that are to be issued by the application component on the atomic register with index 0. Here is an example: `W5:R:W6:R:D200:W3:D100:R`. This means that the application component will issue a Write(5) operation on register 0. Upon completion of this operation, it will issue a Read operation on register 0. Upon completion of this operation, it will issue a Write(6) operation on register 0. Upon completion of this operation, it will do nothing (sleep/timeout) for 200 milliseconds and then it will issue a Write(3) operation on register 0. Upon completion of this operation, it will again wait for 100 milliseconds and issue a Read operation on register 0.

You use the same InitEvent to pass to the components the set of all processes (Π), or the number of processes, respectively. Note that the algorithms assume that each process knows and can communicate with all other processes in Π. This corresponds to a fully connected topology.

These RIWC algorithm, relying on the PFD, assumes that all processes are started within γ[1] milliseconds. To start all your processes at once, you can use the following commands on Windows (cmd) and Linux (bash) respectively:

```
FOR /L %G IN (0, 1, 5) DO start java Assignment4 topology.xml %G ...
for ((i=0;i<=5;i++)) \
   do java Assignment4 topology.xml $i ... > p$i.log 2>&1 & done
```

Replace 5 with your number of processes minus one. For Linux, you can inspect the output of your processes in the `p*.log` files.

Implement the RIWC and RIWCM components and experiment with them as instructed in the following exercises. Describe your experiments in a written report. For each exercise include the topology descriptors used, and explain the behavior that you observe.

You need to read Sections 4.1-4.4 of the textbook, in order to fully understand the ideas behind the two algorithms. These algorithms implement arrays of atomic registers and thus are parameterized by the size of the array. Specify this parameter in a configuration file, `riwc.properties`, that you use to initialize both atomic register components. Here is an example `riwc.properties` file:

[1] the interval between sending two consecutive heartbeats in PFD.

```
capacity = 10        # number of registers in the shared memory
```

The assignment is due on February 27th. You have to send your source code and written report by email before the next tutorial session. During the tutorial session you will present the assignment on a given topology description. You can work in groups of maximum 2 students. Be prepared to answer questions about your process's system architecture and explain the behavior of the algorithms. Any questions are welcome on the mailing list.

Algorithm 1 Read-Impose Write-Consult (part 1)

Implements:
    (N,N)AtomicRegister (nn-areg).

Uses:
    BestEffortBroadcast (beb);
    PerfectPointToPointLinks (pp2p);
    PerfectFailureDetector ($\mathcal{P}$).

```
upon event ⟨ Init ⟩ do
    correct := Π;
    i := rank(self);
    for all r do
        writeSet[r] := ∅;
        reading[r] := false;
        reqid[r] := 0;
        readval[r] := 0;
        v[r] := 0;
        ts[r] := 0;
        mrank[r] := 0;
    end for
end event

upon event ⟨ crash | p_i ⟩ do
    correct := correct \ {p_i};
end event

upon event ⟨ nn-aRegRead | r ⟩ do
    reqid[r] := reqid[r]+1;
    reading[r] := true;
    writeSet[r] := ∅;
    readval[r] := v[r];
    trigger ⟨bebBroadcast | [WRITE, r, reqid[r], (ts[r],mrank[r]), v[r]]⟩;
end event

upon event ⟨ nn-aRegWrite | r, val ⟩ do
    reqid[r] := reqid[r]+1;
    writeSet[r] := ∅;
    trigger ⟨ bebBroadcast | [WRITE, r, reqid[r], (ts[r]+1, i), val] ⟩;
end event
```

Algorithm 2 Read-Impose Write-Consult (part 2)

upon event ⟨ *bebDeliver* | p_j, [WRITE, r, id, (t, j), *val*] ⟩ **do**
    if (t, j) > (ts[r], mrank[r]) **then**
        v[r] := *val*;
        ts[r] := t;
        mrank[r] := j;
    end if
    trigger ⟨*pp2pSend* | p_j, [ACK, r, id] ⟩;
end event

upon event ⟨ *pp2pDeliver* | p_j, [ACK, r, id] ⟩ **do**
    if (id = reqid[r]) **then**
        writeSet[r] := writeSet[r] $\cup$ {p_j};
    end if
end event

upon exists r **such that** correct $\subseteq$ writeSet[r] **do**
    if (reading[r] = true) **then**
        reading[r] := false;
        trigger ⟨*nn-aRegReadReturn* | r, readval[r] ⟩;
    else
        trigger ⟨*nn-aRegWriteReturn* | r ⟩;
    end if
end

Algorithm 3 Read-Impose Write-Consult-Majority (part 1)

Implements:

(*N*,*N*)AtomicRegister (nn-areg).

Uses:

BestEffortBroadcast (beb);

PerfectPointToPointLinks (pp2p).

```
upon event ⟨ Init ⟩ do
    i := rank(self);
    for all r do
        writeSet[r] := ∅;
        readSet[r] := ∅;
        reading[r] := false;
        reqid[r] := 0;
        v[r] := 0;
        ts[r] := 0;
        mrank[r] := 0;
    end for
end event

upon event ⟨ nn-aRegRead | r ⟩ do
    reqid[r] := reqid[r]+1;
    reading[r] := true;
    readSet[r] := ∅;
    writeSet[r] := ∅;
    trigger ⟨ bebBroadcast | [READ, r, reqid[r]] ⟩;
end event

upon event ⟨ nn-aRegWrite | r, val ⟩ do
    reqid[r] := reqid[r]+1;
    writeval[r] := val;
    readSet[r] := ∅;
    writeSet[r] := ∅;
    trigger ⟨ bebBroadcast | [READ, r, reqid[r]] ⟩;
end event

upon event ⟨ bebDeliver | p_j, [READ, r, id] ⟩ do
    trigger ⟨pp2pSend | p_j, [READVAL, r, id, (ts[r], mrank[r]), v[r]] ⟩;
end event
```

Algorithm 4 Read-Impose Write-Consult-Majority (part 2)

upon event ⟨ *pp2pDeliver* | p_j, [ReadVal, r, id, (t, rk), val] ⟩ **do**
  if (id = reqid[r]) **then**
    readSet[r] := readSet[r] $\cup$ {((t, rk), val)};
  end if
end event

upon exists r **such that** |readSet[r]| $> N/2$ **do**
  ((t, rk), v) := $highest$(readSet[r]);
  readval[r] := v;
  if (reading[r] = true) **then**
    trigger ⟨*bebBroadcast* | [Write, r, reqid[r], (t, rk), readval[r]]⟩;
  else
    trigger ⟨*bebBroadcast* | [Write, r, reqid[r], (t+1,i), ~~readval~~[r]]⟩;
  end if
end

upon event ⟨ *bebDeliver* | p_j, [Write, r, id, (t, j), val] ⟩ **do**
  if (t, j) > (ts[r], mrank[r]) **then**
    v[r] := val;
    ts[r] := t;
    mrank[r] := j;
  end if
  trigger ⟨*pp2pSend* | p_j, [Ack, r, id] ⟩;
end event

upon event ⟨ *pp2pDeliver* | p_j, [Ack, r, id] ⟩ **do**
  if (id = reqid[r]) **then**
    writeSet[r] := writeSet[r] $\cup$ {p_j};
  end if
end event

upon exists r **such that** |writeSet[r]| $> N/2$ **do**
  if (reading[r] = true) **then**
    reading[r] := false;
    trigger ⟨*nn-aRegReadReturn* | r, readval[r] ⟩;
  else
    trigger ⟨*nn-aRegWriteReturn* | r ⟩;
  end if
end