ID2203 Tutorial 3 - Shared Memory

Cosmin Arad Tallat M. Shafaat
icarad(@)kth.se tallat(@)kth.se

Handout: February 19, 2010
February 26, 2010

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 Kompics 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 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: AtomicRegister-ReadEvent, AtomicRegisterReadReturnEvent, AtomicRegisterWriteEvent, AtomicRegisterWriteReturnEvent and the messages that they exchange: ReadMessage, WriteMessage, AckMessage, and ReadValueMessage.

You will slightly change the application component from the previous 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.

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. For some exercises, you are required to start a process later than others. This is done by specifying the delay in the command of a scenario. For instance, given the following scenario:

```
Scenario scenario1 = new Scenario(AssignmentXGroupY.class) {
   {
      command(1, "");
      command(2, "");
      command(3, "", 1000);
   }
};
```

The system will launch process 1 and 2, while process 3 will start executing after 1000 ms.

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, which you have to use as a parameter to the algorithms.

The assignment is due on February 26th. 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 forum.

Exercise 1 Verify the termination and atomicity properties of RIWC and RIWCM. Use a topology with 3 processes and 3 bidirectional links with the same latency (say 1000ms) and execute RIWC and RIWCM. Use the following ops: process 1: `D30000`, process 2: `D500:W4:D25000` and process 3: `D10000:R`. Here you have sequential operations and you should check that the read returns the last value written.

Exercise 2 Use a topology with 3 processes and 3 bidirectional links with the same latency (say 1000ms) and execute RIWCM. Start process 1 with ops: `D500:W5:R:D5000:R:D30000` and process 2 with ops: `D500:W6:R:D5000:R:D30000`, and don't start process 3 yet. This is a special case of failure called "initialy dead processes". Create your scenario such that when processes 1 and 2 begin to wait for 30 seconds, process 3 starts with ops: `D500:R:D500:R:D10000`.

What is the value read by process 3? Explain why. Can RIWCM be used in a fail-recovery model?

Exercise 3 Using the topology given below, or a similar one, execute RIWCM for each possible starting order of the processes: 123, 132, 213, 231, 312, 321, with the following operations for process i: `D500:Wi:R:D500:R:D8000`. Order of processes means that the 3 processes should be started with a small delay (e.g. 100ms) between any 2 consecutive processes. For instance, for order 123, you can use the following scenario:

```
Scenario scenario1 = new Scenario(AssignmentXGroupY.class) {
   {
      command(1, "");
      command(2, "", 100);
      command(3, "", 200);
   }
};
```

Give the values returned by the two read operations in each process and give a linearization of all the register operations (read and write from all processes).

```
Topology topologyEx3 = new Topology() {
   {
      node(1, "127.0.0.1", 22031);
      node(2, "127.0.0.1", 22032);
      node(3, "127.0.0.1", 22033);
```

```
        link(1, 2, 1000, 0).bidirectional();
        link(1, 3, 2000, 0).bidirectional();
        link(2, 3, 1750, 0).bidirectional();
    }
};
```

Exercise 4 Change the latencies in the previous topology, and possibly the waiting values in ops so that you obtain 3 distinct linearizations so that in each linearization, the write of value i is the last of the 3 write operations.

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[$r$] ∪ {$p_j$};
    end if
end event

upon exists $r$ such that correct ⊆ 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
```