Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

HS 2012 Prof. R. Wattenhofer / B. Keller

Distributed Systems Part II

Solution to Exercise Sheet 1

1 An asynchronous Riddle

a) The crucial idea is to select one prisoner as a leader. The leader will turn the switch off, whenever he enters the room and the switch is on. All other prisoner will turn the switch on exactly once. So a prisoner who enters the room looks at the switch. If the switch is off and the prisoner has never turned it on before, he will turn the switch on. If the switch is already on or the prisoner already did turn the switch on during an earlier visit, he leaves the switch as it was. The leader counts how many times he turns the switch off. If the leader counted 99 times he can declare "We all visited the switch room at least once". Because he knows, that each of the other 99 prisoners has turned the switch on and he himself has been in the room as well.

b) If the initial position of the switch is unknown, the above protocol cannot be used, since the leader may has miscount by one. However, this can easily be fixed. If each prisoner turns the switch on exactly twice, the leader can, after having counted $2 \cdot 99 = 198$ times.

2 Communication Models

Some ideas are:

Delay There is no delay with shared memory, if one process writes the other processes can read immediately. With messsage passing delay can happen (not necessarily in every model), different messages may even have different delays.

Overriding With shared memory if a process writes to a register, another process may override the value before anyone could read the register. In message passing this cannot happen. On the other hand messages may be lost, or the inbox buffer of a process may overflow, leading to similar results.

Consistency With message passing several message may be sent at the same time, and the order of arriving message may be messed up. With shared memory the value of a register is always the value that was written last.

3 Communication without Computers

Also remember the discussions during the exercise session as this list does not consider all aspects.

Postcard Clearly message passing. Messages may be lost, the order may be inconsistent and the inbox may overflow.

Two people speaking Shared memory: one can speak (=write), the other can listen (=read). Both speaking does not work, both listening neither.

Skype This could be modeled with message passing or with shared memory.

Message passing: the text to send is a message, it cannot be lost and their order is consistent. However, it would be stored on the sender‘s site until the receiver is online.

Shared memory: there could also be an “inbox” register for each participant. The sender writes into the others inbox register. The receiver clears its register once it has seen the new text, the sender could then write into the register again.

Many people speaking Message passing: everyone is connected with everyone. If speaking message are sent not only to the intended listener(s) but also randomly to other people. If someone speaks loud, more messages are sent. The inbox of any person has limited size; once it is full, arriving messages begin to override older messages. This models the fact that one cannot listen to many speakers at the same time.

4 Consensus with an n-Register

We require 6 registers. We call the first three registers R_0, R_1 and R_2. To the other three registers we give the names $R_{\{0,1\}}$, $R_{\{0,2\}}$ and $R_{\{1,2\}}$. The goal is to find the *fastest* process and take its input value as decision. In words, the protocol works as follows:

In a single step process i writes its id into R_i and into $R_{\{i,j\}}$ for $i \neq j$.
It then checks for all $i \neq j$ whether process i was faster than process j:
  If $R_{\{i,j\}} = -1$ then neither i nor j have yet done anything.
  Otherwise, if $R_i = -1$ then process j must be faster than i.
  Otherwise, if $R_j = -1$ then process i must be faster than j.
  Otherwise $R_{\{i,j\}}$ holds the id of the process which was slower.
With all this information, a process can calculate which process must have been the fastest.

Solution in pseudo code:

```
initialize(){

  // R are the shared registers
  R[] = [-1, -1, -1, -1, -1, -1];
  // the input, an array of length 3
  input[] = [random(), random(), random()];
}

decide(){

  id = this.getThreadId();
  // the identifiers of the other processes
  others = [ {0,1,2} without {id} ];

  // atomically write three registers
  write( R[id] = id, R[id,others[0]] = id, R[id,others[1]] = id );
  // pairwise comparison of process-speed
  fastest01 = faster( 0, 1, id );
  fastest02 = faster( 0, 2, id );
  fastest12 = faster( 1, 2, id );

  // find the process which is faster than all the others
  score[] = [0, 0, 0];
  score[ fastest01 ] = score[ fastest01 ]+1;
  score[ fastest02 ] = score[ fastest02 ]+1;
  score[ fastest12 ] = score[ fastest12 ]+1;
  winner = max( score );

  if( count[0] == winner )
    decision = input[0]
  else if( count[1] == winner )
    decision = input[1];
  else // count[2] == winner
    decision = input[2];
}

faster( i, j, id ){

  somethingHappened = true;
  while (somethingHappened){
    // We need to assure, that we read a consistent memory state
    // This is still wait-free, this loop is traversed at most 3 times
    rij = R[i,j]; ri= R[i]; rj = R[j];
    rijSecond = R[i,j]; riSecond = R[i]; rjSecond = R[j];
    if(rij == rijSecond && ri == riSecond && rj == rjSecond) {
      somethingHappened = false;
    }
  }
  if( rij == -1 ){ // neither of i or j yet started, I am faster than both
    return id;
  }
  else{
    if( ri == -1 ){
    // i did not yet start, hence j must be faster
    return j;
    }
    if( rj == -1 ){
      // j did not yet start, hence i must be faster
      return i;
    }
    if( rij == i ){
      // value written by j was overridden by i
      return j;
    }
    else{ // rj == j
      return i;
    }
  }
}
```

5 Consensus for two Processes

The protocol works and achieves consensus. Let's have a closer look at the code. The loop is never executed more than twice and we can easily get rid of it, this also eliminates the variable `decisionMade`. The simplified version looks like this:

```
// making the decision
decide(){

  // the id of this process , 0 or 1
  id = this.getThreadId();

  ////////
  // a*
  ////////

  value = s;
  if( value == '?' ){
    s = input[ id ];
  }
  value = s;

  ////////
  // b*
  ////////

  if( value != input[ id ] ){

    ////////
    // c1*
    ////////

    decision = value;
  }
  else{

    ////////
    // c2*
    ////////

    if( i.fetchAndInc() == 1 ){
      decision = input[ 1-id ];
    }
    else{
      decision = input[ id ];
    }
  }
}
```

All processes will pass **a***, **b***, and either **c1*** or **c2***. If we look at **a*** and `input[0] == input[1]`, then the protocol trivially reaches consensus. For us only the cases where the inputs differ are interesting. Out of symmetry it is enough to show that the protocol succeeds if `input[0] == 0` and `input[1] == 1`. When reaching **b*** either both processes have read the same values or they did not.

- In the case of $\texttt{value}_0$ `==` $\texttt{value}_1$ one process will enter the branch with **c1***, the other the branch with **c2***. The one passing through **c1*** will change its decision, the other passing through **c2*** will not hence both processes end with the same decision.
- In the case of $\texttt{value}_0$ `!=` $\texttt{value}_1$ both processes have read their own input. In this case both processes pass **c2***. The second one will change its decision because `i.fetchAndInc()` returns 1, the other one will not, hence both processes end with the same decision.
- The case where $\texttt{value}_0$ `!=` $\texttt{value}_1$ and both processes have $\texttt{value}_i$ `!=` $\texttt{input}[i]$ never happens. We prove this with an execution tree for the code between **a*** and **b***.

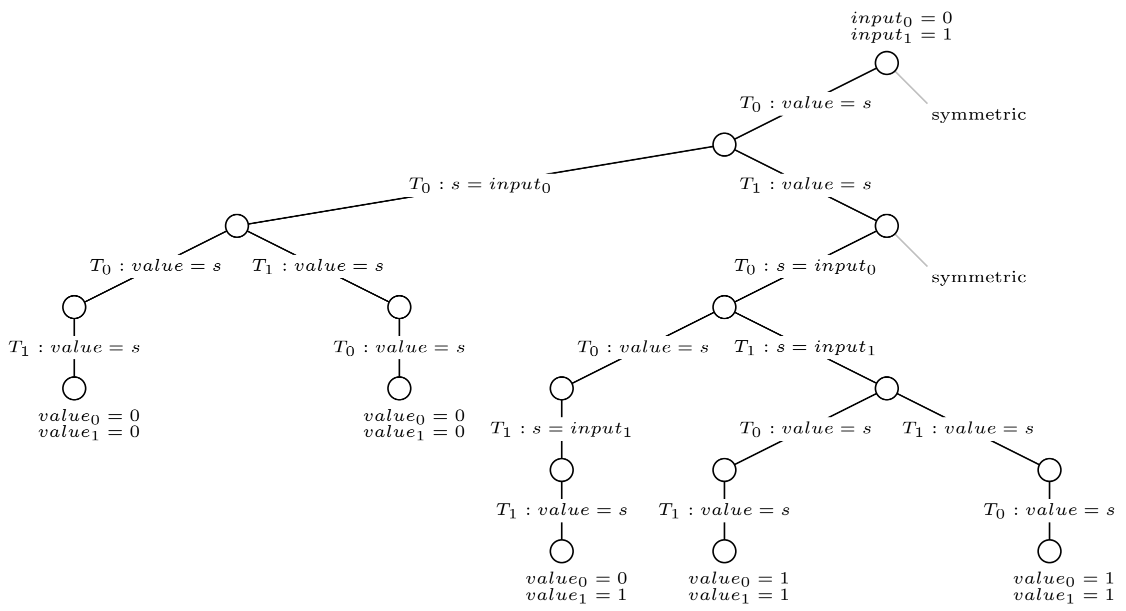

Figure 1: Execution tree for the code between **a*** and **b***.