ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

HS 2014 Prof. R. Wattenhofer

Distributed Systems Part II

Solution to Exercise Sheet 2

1 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.

2 Consensus with the Aid of a Wall

a) The Algorithm looks like this:

Choose the color of the place you want to meet.
Go to the Painter and instruct him to paint the wall in the corresponding color
Look at the "before and after" picture which you get from the painter.
if {the wall was white before} {
  the meeting place is the one you have chosen
} **else** {
  the meeting place is the place according to the "before color"
}

b) No. What Alice and Bob can do, is in a sense the same as the RMW-primitive swap. As swap is overwriting its consensus number is two. So there is no way to ensure that more than two persons can meet at the same place.

c) If the wall is in front of the painters' shop it is basically the same as the RMW-primitive Compare and Swap (because you would see if the painter is already painting, or if someone is in the shop and instructing the painter to paint) which has consensus number ∞. This means, that infinitely many persons could meet. The Algorithm would look like this:

Choose the color of the place you want to meet.
Go to the Painter
Look at the wall in front of the painters' shop
if {the wall is white } {
  enter the shop and instruct the painter to paint the wall in your color
  the meeting place is the one you have chosen
} **else** {
  the meeting place is the place according to the color of the wall
}

3 Consensus through "Fetch and Multiply"

I would tell him, that this is not possible. The method "Fetch and Multiply" he uses is commutative, therefore the consensus number of his algorithm cannot exceed two.

4 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 $\texttt{value}_i$ `!=` `input[`i`]` never happens. We prove this with an execution tree for the code between **a*** and **b***.

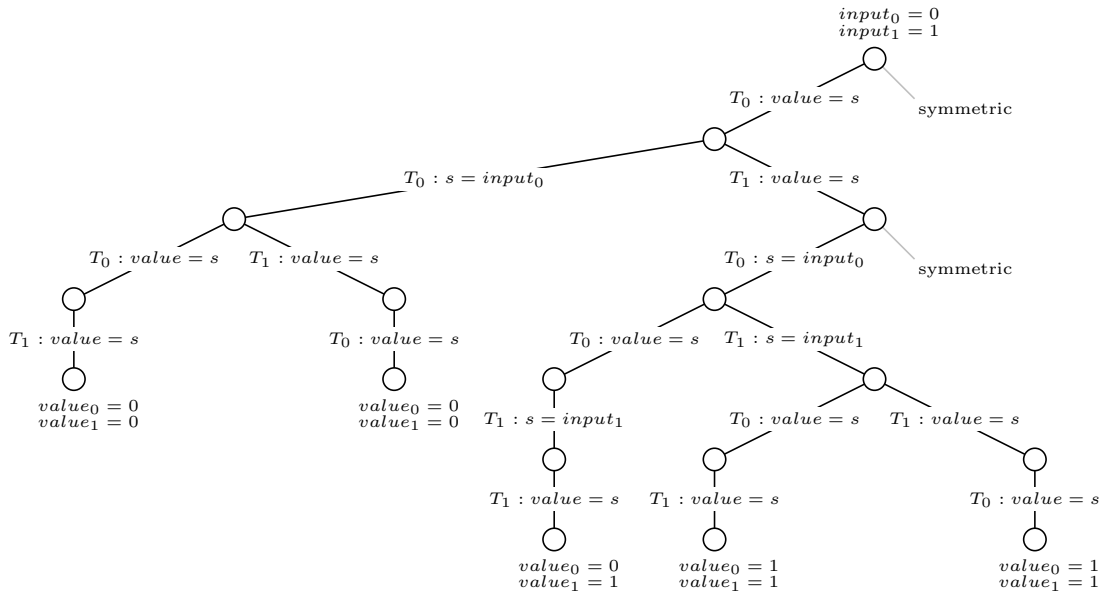

Figure 1: Execution tree for the code between **a*** and **b***.