Royal Institute of Technology

MSc. Software Engineering of Distributed Systems

ID2203 Distributed Systems Advanced Course

Homework 2

Andrei Shumanski
andreish@kth.se
0046707761992

Trigonakis Vasileios
vtri@kth.se
0046707694420

Stockholm 2010

TABLE OF CONTENTS

Exercise 2 4

Algorithm 4

Algorithm Explanation 5

Book's algorithm criticism & our algorithm's improvements 5

Criticism of our algorithm 6

Question 5 7

5(a) 7

5(b) 8

5(c) 10

5(d) 12

Exercise 2 13

Question 1 13

EXERCISE 2

ALGORITHM

Implements:

ProbabilisticBroadcast (pb).

Uses:

FairLossPointToPointLinks(flp2p);
UnreliableBroadcast(un).

```
upon event <Init> do
        forall Pi ∈ Π do
                delivered[Pi] := 0;
                missing[Pi] := 0;
        lsn := 0; stored := 0;

procedure gossip (msg) is
        forall t ∈ pick-targets (fanout) do
        trigger <flp2pSend | t, msg>;

upon event <pbBroadcast | m> do
        lsn := lsn+1; trigger <unBroadcast | [Data, self, m, lsn]>;

upon event < unDeliver | Pi, [DATA, Sm, m, SNm]> do
        if (store-thresold < random()) then
                stored := stored ∪ { [DATA, Sm, m, SNm] };
        if (SNm >= delivered[Sm] + 1)  then
                trigger < pbDeliver | Sm, m >;
                forall seqnb ∈ [delivered[Sm] + 1, SNm - 1] do
                        gossip ([REQUEST, self, Sm, seqnb, maxrounds – 1]);
                        missing[Pi] := missing[Pi] ∪  seqnb;
                delivered[Sm] := SNm;
        else if (SNm ∈missing[Sm]) then
              missing[Sm] := missing[Sm] \ SNm;
                trigger < pbDeliver | Sm, m >;

upon event < flp2pDeliver | Pj, [REQUEST, Pi, Sm, SNm, r] > do
        if ([DATA, Sm, m, SNm] ∈ stored) then
                trigger < flp2pSend | Pi, [DATA, Sm, m, SNm] >;
        else if (r > 0) then
                gossip ([REQUEST, Pi, Sm, SNm, r – 1]);

upon event < flp2pDeliver | Pj, [DATA, Sm, m, SNm]> do
        if (SNm ∈ missing[Sm]) then
                missing[Sm] := missing[Sm] \ SNm;
              trigger < pbDeliver | Sm, m >;
```

ALGORITHM EXPLANATION

1. Lines 1-5: initialization of the structures that it uses. It does not use a pending array, but instead it is using an array of sets of integers where a node can keep the missing messages from the other nodes. As it appears, this structure was anyway needed by the book's algorithm, if we wanted to increase the probability of receiving messages that were delayed more than the timeout that the algorithm uses.
2. Lines 7-9: the same as the book's algorithm. Note: when a node is forwarding the request of another node, it should not pick the requester node
3. Lines 11-12: the same as the book's algorithm
4. Lines 14-25: handling of an message coming from the unreliable broadcast. Again, randomly it may store a received message or not. Then (line 17), if the node receives a new message (message with higher sequence number from the sending node), then it **immediately** delivers it, starts a gossip for each "missing" message and place the missing node to the missing structure. The changes that we made, allow us not to use timer for our algorithm to work. This is a big improvement, because we no more any timing assumptions, so our algorithm can work (almost) equally well in an asynchronous system. Finally, if we receive a message that belongs to the missing structure (the case were due to the network two messages are coming with the wrong order) then we deliver it and remove it from the missing list.
5. Lines 27-31: the same as the book's algorithm
6. Lines 33-36: when we receive a deliver from the flp2p link (coming from a gossip response), if it belongs to the missing we remove it and deliver it, else we simple ignore it.

BOOK'S ALGORITHM CRITICISM & OUR ALGORITHM'S IMPROVEMENTS

The algorithm of the book had some severe logic and performance problems. Here they are, with an explanation how we did solve them:

1. In order to call the deliver-pending procedure, it should receive a message from gossip with a s/n next to the one that it had already delivered. That means that if a message is totally lost (we have a probabilistic broadcast anyway) then the algorithm will block.
   Solution: we deliver messages immediately, if the should be delivered, else we skip them

2. Combined with the problem no. 1, when it receive a timeout about a missing message, it does not handle the case where we didn't receive the proper "message". The algorithm is somehow "stubborn" and does not accepts to name some messages as lost and continue running.
   Solution: we do not use timeout at all

3. If due to network delays, two messages from the same node are unDelivered with the wrong order (i.e. first message with s/n 9 and then message with s/n 8, both from process 2), then the second one is dropped by the algorithm. Instead of this it could deliver it and resolve some pending messages faster!
   Solution: we handle the case when two messages come from in reverse order by checking into the missing list for the second message. If it exists, we deliver it and remove it from the missing list.

4. Except from the locking issues, the book's algorithm (tries to) implements ordered broadcast. As it is obvious, this a stronger assumption than what we need that cannot be implemented by our probabilistic algorithm. Even if it was possible, this strong assumption delays the delivery of the received messages for no reason.
   Solution: we deliver the messages as soon as we receive them

5. When the algorithm receives a gossip data message, if it not the "correct" message, it is always added in the pending set. This means that pending set can contain duplicate pending messages, which is inefficient.
   Solution: we add the missing messages in the list, only in one place

CRITICISM OF OUR ALGORITHM

Our algorithm is correct, faster and simpler than the book's one. Also, because of the non-use of timeouts, it is more robust and easier to work over an asynchronous system. The only issue that we identified is the growing data structures. Both stored and missing data structures are getting larger as the time passes.

Solution:

- A simple solution that can be applied is to restrict the message of the structures and shift the oldest values out of them. For example, if we say that we mostly store 40 messages, when the structure gets full, the oldest item is thrown for a new one to take place. If we use this technique, we can also implement an interesting variation of the algorithm. We could store all the new messages coming, throwing the older ones. This could help us increase the chance of delivery of a message because new messages have more chance to be asked than older one. Of course this technique would be usable only with a correct combination of message delivery frequency, data structure size and network delays.
- A more complex solution could be to have dynamic structures that increase (and decrease) their size according to the node's activity, assuming the we using the previous solution also. This could guaranty that if the node is under stress due to many losses, it will not lose messages due to full data structures, but in the common case, its structures are not big, so they do not affect its performance.

QUESTION 5

Lazy probabilistic broadcast depends on three parameters: fanout, store-threshold and maxrounds. Discuss in your report how do these parameters affect the broadcast. In your report, describe executions by varying the values of these parameters and the topology characteristics that lead to the following scenarios.

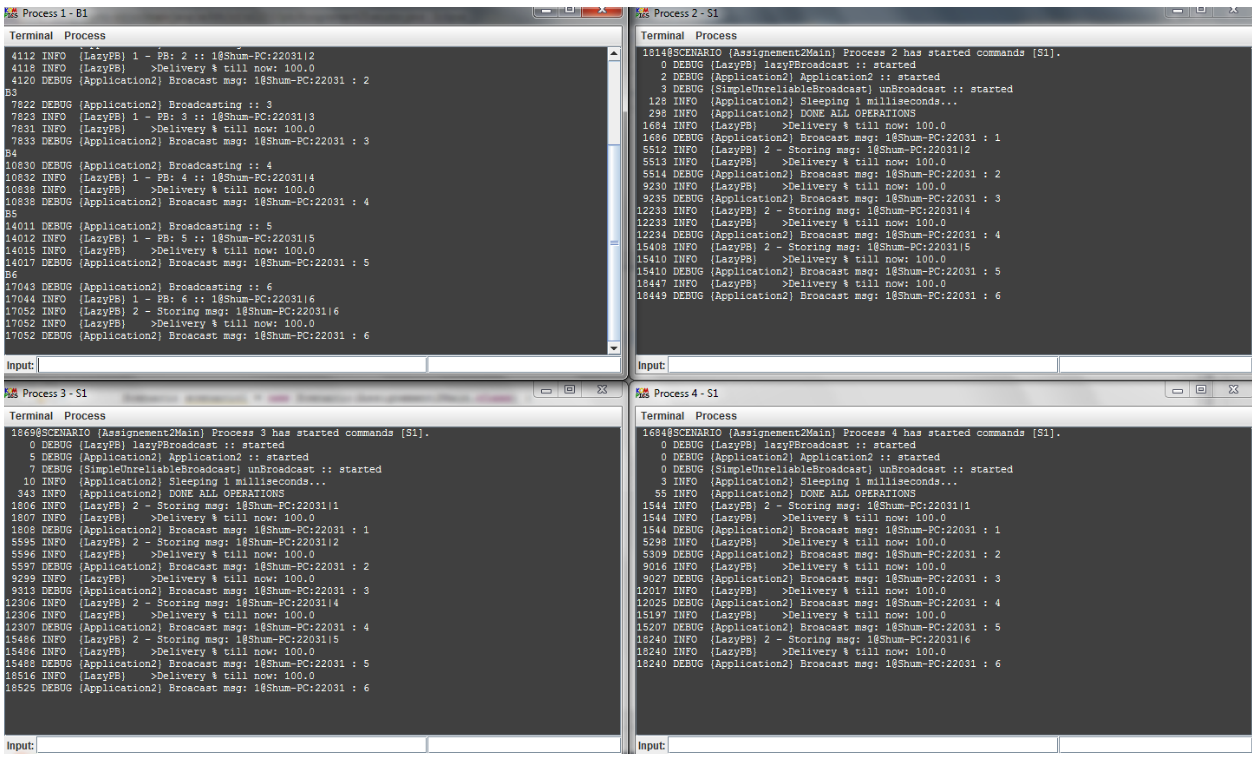
5(A)

No message is lost.

We set lost of the messages in the network to 0:

```
defaultLinks(1000, 0);
```

In this case all messages are delivered by unreliable broadcast so gossip is never executed:



So the result doesn't depend on *fanout*, *store-threshold* and *maxrounds* parameters because they affect only gossip phase of the algorithm. All nodes deliver all messages during unreliable broadcast and we can use only it.

5(B)

A broadcasted message is lost in the unreliable broadcast but recovered by gossip for some node *p*.

To model lost of some messages during unreliable broadcast we made network lost 50%, this is a lot but we can see how algorithm works in extreme cases:

```
defaultLinks(1000, 0.5);
```

As we use fair-loss link we cannot GUARANTEE that all message will be recovered as we can lose all gossip messages.

If we want to increase chances to recover message at gossip phase we should increase *store-threshold* to be sure that if some node delivers the message it will be able to send it during the gossip. We also should increase *fanout* and *maxrounds* number of fanouts increases chance of finding the node with stored message and maxrounds increases chance to find lost message from the nodes which are not connected as fanouts.

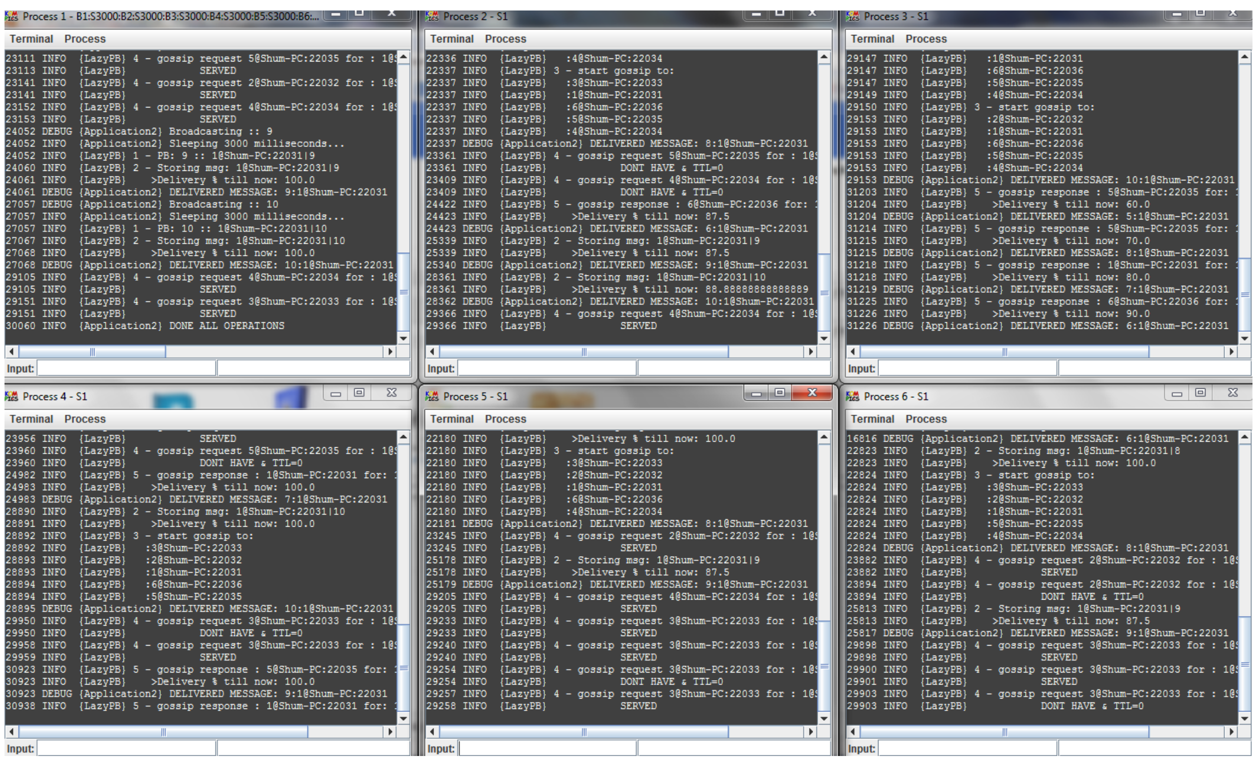
In our experiment we used 6 nodes because with greater number of nodes it becomes difficult to analyze outputs.

```
private static final double storeTreshold = 1;

private static final int fanouts = 5;

private static final int ttl = 1;
```

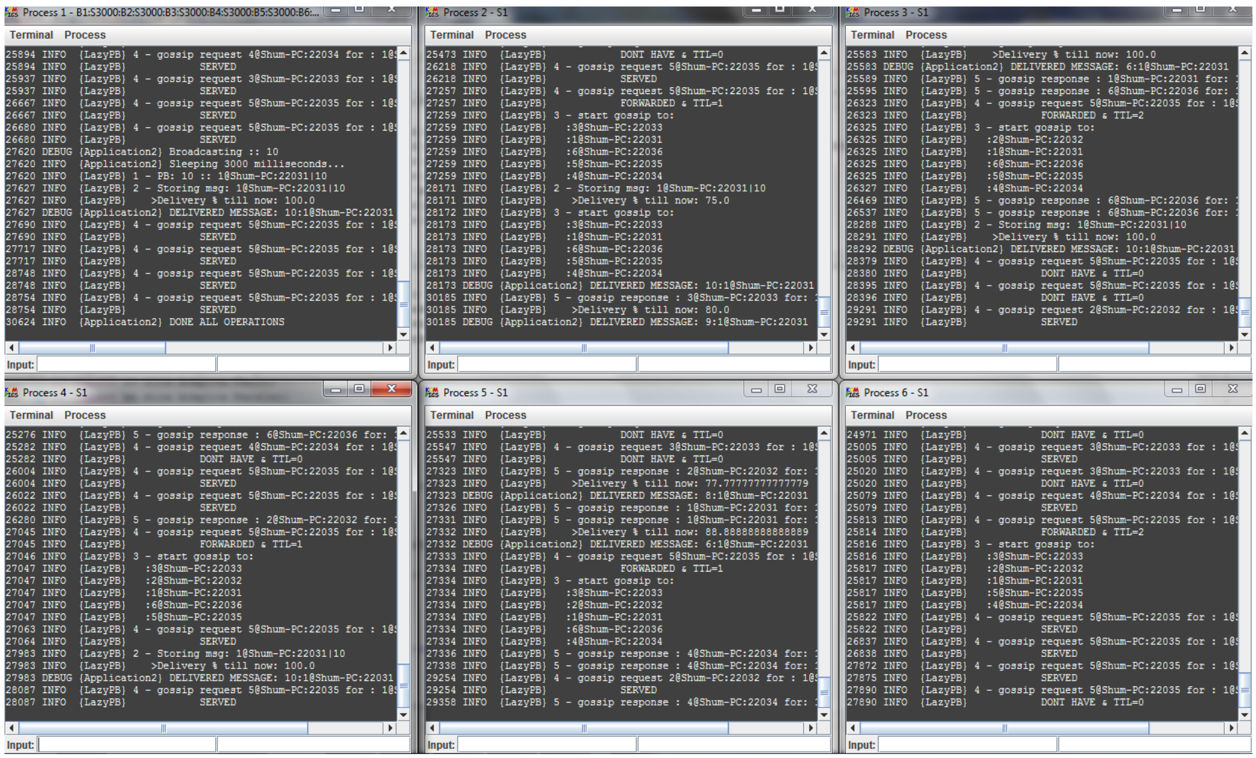
In this configuration all nodes are connected to all others as fanouts and store-treshhold is 100% so we should lose messages only if it was lost during the unreliable broadcast and gossip phase.



2 nodes delivered 100% of messages and 4 nodes delivered ~90% of messages.

Let’s increase maxrounds to 2, it should increase the chances:

```
private static final double storeTreshold = 1;
private static final int fanouts = 5;
private static final int ttl = 3;
```



4 nodes delivered 100% of messages, one node – 80% and one node – 90%, so delivery rate really increased.

5(C)

A broadcasted message is lost such that although it is stored on some node(s) in the network, a node p missed it in the unreliable broadcast and furthermore, p could not retrieve it via gossiping as well.

To modes such situation we keep the network lost rate 50%:

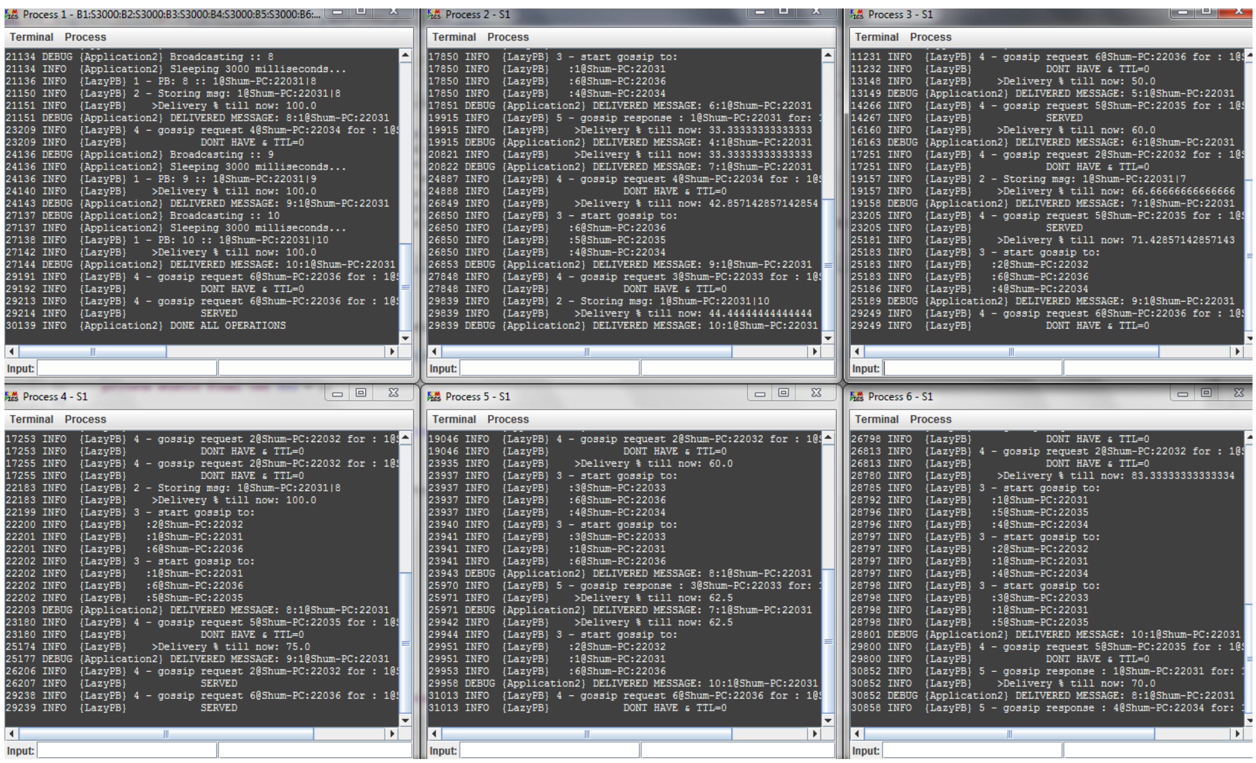
```
defaultLinks(1000, 0.5);
```

Now not all nodes that deliver message should store it, so we decrease *store-threshold* to 30% and we also decrease *fanouts* to 3 to make chances of gossip failure higher:

```
private static final double storeTreshold = 0.3;

private static final int fanouts = 3;

private static final int ttl = 1;
```



Results for nodes are:

Node	Delivery rate
1	100%
2	44%
3	66%
4	75%
5	62%
6	70%

Node 1 has high delivery rate because it sends broadcast messages and delivers all of them so we can exclude it from comparison. We can see that comparing to previous test chances to fail gossip increased because some messages were not stored.

5(D)

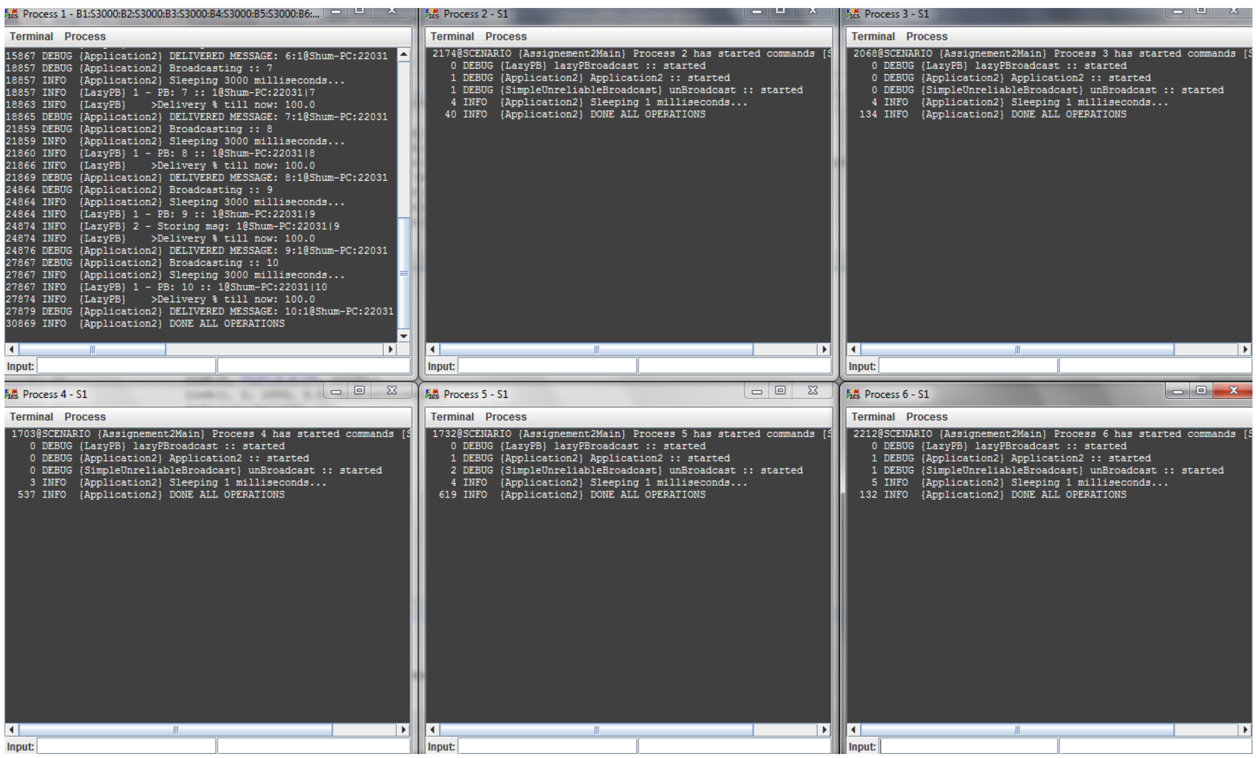
A broadcasted message, after being delivered by some node(s) and missed by a node p, is completely lost such that p can never retrieve it through gossiping.

We can guarantee that message will be never retrieved through gossiping by setting to 0 configuration parameters but according to PDF we can't do that.

Another way to create such execution is to set network lost to 100%

```
defaultLinks(1000, 1);
```

In this case only sending node delivers the message and all other nodes p do not deliver it.



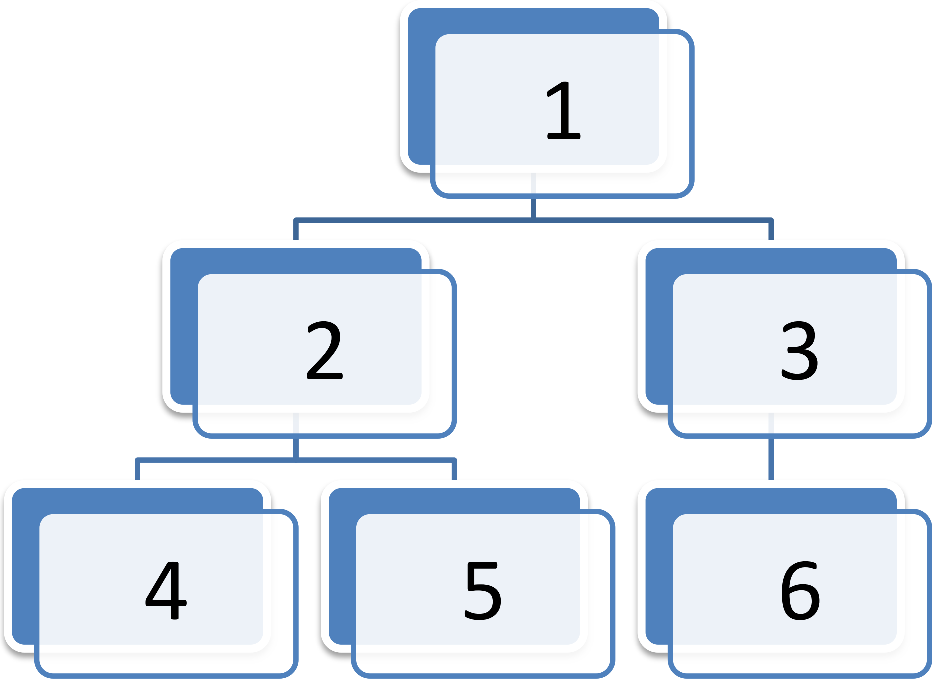
The first node delivers 100% messages and all others deliver nothing.

EXERCISE 2

QUESTION 1

Do you think the lazy probabilistic broadcast algorithm will work in such a topology? What implications will the afore-mentioned topology have on the algorithm?

No, the lazy probabilistic broadcast will not work in not fully connected topology.



The problem is that unreliable broadcast described in Algorithm 1 sends messages only to neighbors so if node 1 broadcasts something it is delivered only by nodes 2 and 3. Nodes 4, 5, and 6 never deliver broadcast messages from 1 and they do not know if they missed some message from node 1 or node 1 never send any messages so they cant start gossip.