

Software Engineering of Distributed Systems, KTH

Distributed Systems Advanced Homework 3

ReliableBroadcast, UniformReliableBroadcast and ProbabilisticBroadcast

Exercise 1. Reliable Broadcast

We record the number of the same message which we received. If **this number is greater than current correct nodes' number** then we affirmative that the initial source of this message has already crashed and all other nodes has forward this message to me. So I will never get this message again. And there is no need to store this message in deliver list to filter.

Principle:

A node will get a same message by no more than the number of current correct nodes plus one.

Scenario 1: (focus on p1)

A node (p1) gets a message *m* from *m*'s initial node (p0).

When it detects p0 crashed. It will broadcast *m* to all lived nodes include himself. And others will eventually do the same thing. Then p1 gets *m* by the number of current correct nodes since p0 crashed. And p1 get the first *m* from p0, *m*'s initial node. So the number of *m* which p1 get is the number of current correct nodes plus one. P1 will never get *m* again, so it can do the garbage collection to remove *m* from delivered list.

Scenario 2: (focus on p1)

1. Set of all node = {p0, p1, p2, p3, p4}
2. p0 sends *m* to all
3. p0 crashes and failed to send *m* to p1
4. p2, p3, p4 get *m*
5. p2 detects p0 crashed and broadcast *m* to all nodes.
6. p1 gets *m* for the first time from p2 (counter(*m*) = 1, correct.size = 4)
7. p3, p4 detect p0 crash and broadcast *m* to all nodes.
8. p1 gets *m* from p3, p4 (counter(*m*) = 3, correct.size = 4)
9. p3, p4 crash.
10. p2 crashes
11. p1 detects p2 crashed and broadcast *m* to all nodes.
12. p1 gets *m* from himself. (counter(*m*) = 4, correct.size = 1)
13. counter(*m*) > correct.size() do garbage collection(*m*)

The code for this algorithm is illustrated below:

Data structure:

```
private Map<SourceMessagePair, Integer> delivered;
```

Algorithm

```
public void handleBebDeliverEvent(BebDeliverEvent bebDeliverEvent) {  
    ...  
    delivered.put(sourceMessagePair, new Integer(delivered.get(sourceMessagePair) + 1));  
  
    if (delivered.get(sourceMessagePair).size() > correct.size()) {  
        delivered.remove(sourceMessagePair);  
        System.out.println("");  
        System.out.println("Garbage collect delivered list, remove message \"  
            + sourceMessagePair.getMessage() + \"\" from Node "  
            + sourceMessagePair.getSource().getId());  
    }  
    ...  
}
```

output Log:

The output logs located in `\Logs\exercise1`

Operation:

- node 0:
 - send "a"
 - send "b"
 - send "c"
 - crash
- node 4:
 - crash (before d(0))
- node 3:
 - crash (after d(4))

Exercise 2: Uniform Reliable Broadcast

All-Ack Uniform Reliable Broadcast implements the uniform version of reliable broadcast in the fail-stop model. In this algorithm, a process delivers a message only when it knows that the message has been delivered by all correct processes. All processes relay the message once they have delivered it.

We update the origin algorithm, so that it garbage collects the messages in pending list, and it only keeps track of the last message sent from each process in the delivered list.

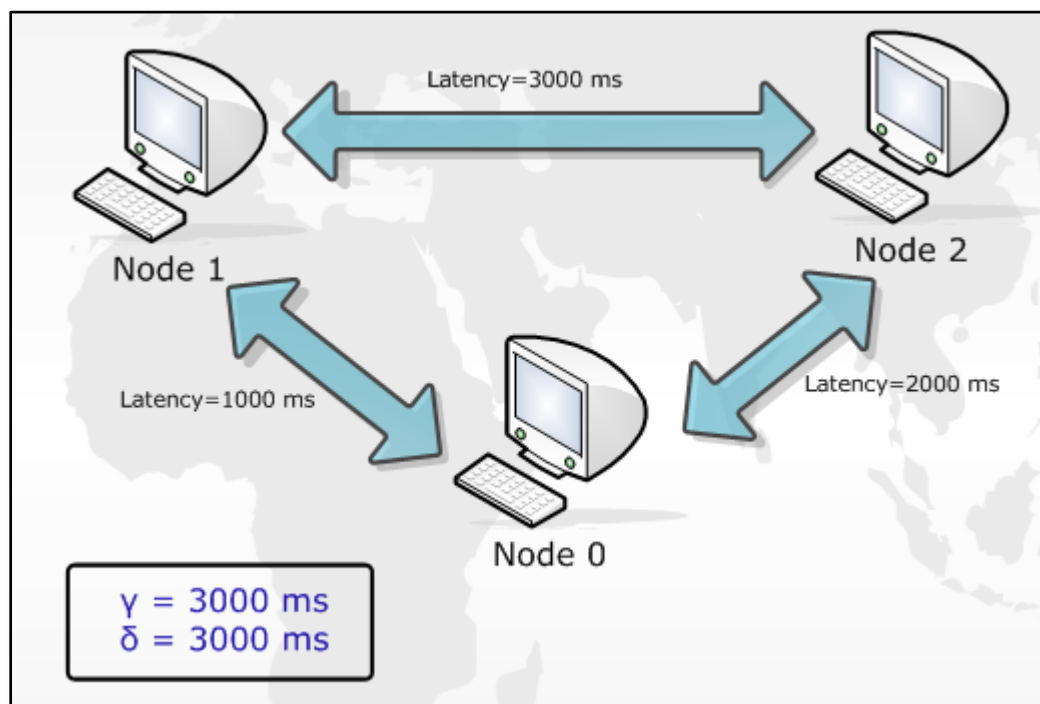
```
private void deliver() {
    for (Map.Entry<NodeReference, Set<SourceMessagePair>> entry : pending.entrySet()) {
        Set<SourceMessagePair> msgs = entry.getValue();
        for (SourceMessagePair msg : msgs) {
            if (canDeliver(msg)) {
                delivered.put(msg.getInitNode(), msg);
                garbageMsg.add(msg);
                component.raiseEvent(new RbDeliverEvent(msg.getTimeStamp(), MessageType.
            }
        }
    }
    if (!garbageMsg.isEmpty()) collectGarbage();
}
```

As shown in the snippet above, the algorithm is very simple, just make the garbage collection happen after the delivery of a reliable broadcast message. As soon as a message is delivered, this message will be removed from the pending list, since it is indeed delivered and we don't have to take care of it anymore in the pending list.

On the other hand, we think the delivered list is not necessary in this algorithm, because what will be delivered are those messages in the pending list, waiting for acknowledgements from other nodes, and each message sent by certain node is unique, and the acknowledgement for this message is sent and only sent once.

The property of best effort broadcast tells us there is neither creation nor duplication of message, so we won't have two same messages or acknowledgement, therefore we don't even have to keep a record of the delivered messages.

We experiment our algorithm in the topology shown below.



Scenario 1:

Node 0 broadcasts three messages: 0, 00, 000, we manually crashes node 0 after it delivers 0.

Node 0

Start RelibaleBroadcast message: 0

Start RelibaleBroadcast message: 00

Start RelibaleBroadcast message: 000

Get a rbDeliverEvent -> msg:0, init:ref://127.0.0.1:13340/0, timeStamp:1203454145750

Pending list :

"000" from Node 0

"00" from Node 0

We manually crashes node 1 after it delivers 0 and 00, even before it detects the crash of node 0.

Node 1

Get a rbDeliverEvent -> msg:0, init:ref://127.0.0.1:13340/0, timeStamp:1203454145750

Pending list :

"000" from Node 0

"00" from Node 0

Get a rbDeliverEvent -> msg:00, init:ref://127.0.0.1:13340/0, timeStamp:1203454146335

Pending list :

"000" from Node 0

We let node 2 run till the end, one can see that it delivers all messages. It must deliver 0 and 00, because node 1 delivers 0 and 00 (though it has crashed), it can deliver 000, since there is no other correct process expect itself.

Node 2

Get a rbDeliverEvent -> msg:0, init:ref://127.0.0.1:13340/0, timeStamp:1203454145750

Pending list :

"000" from Node 0

"00" from Node 0

Get a rbDeliverEvent -> msg:00, init:ref://127.0.0.1:13340/0, timeStamp:1203454146335

Pending list :

"000" from Node 0

Get a rbDeliverEvent -> msg:000, init:ref://127.0.0.1:13340/0, timeStamp:1203454147269

Pending list :

Empty!

Node 2 : discovers that node ref://127.0.0.1:13340/0 crashed!

Node 2 : discovers that node ref://127.0.0.1:13341/1 crashed!

Exercise 3. Study of Lazy Probabilistic Broadcast

The algorithm in the Textbook doesn't consider thread in java.


Every timeout is a single thread, so there is no guarantee for the timeout to happen as the order of raising. So it may happen a $sn = delivered[s] + n$ ($n > 1$) happen before $sn = delivered[s] + 1$. As long as it happens, it will make the whole mechanism stop and waiting for a further message to wake up a new round of request.

The differences between the two algorithms are shown on the follow table:

| | Textbook | Tutorial |
|-----------------------------|---|---|
| Check timeout | Just check the next message's request timeout | Check every request timeout |
| Nil message | N/A | Insert a nil message when the request for that timeout and put it back if receive it in the future. |
| Check pending after timeout | no | yes |
| Deliver | Continuously deliver, stop when meet a gap. | Can continuously deliver message in pending list, if there is a nil in the pending list it can skip is and deliver the further message. |
| Efficient | low | Good |

Example scenario:

Note: the timeout may happen without a certain order.

| | Textbook Algorithm | Tutorial Algorithm |
|--|---|--|
|  | deliver[s] = 0, get 1, deliver[s] = 1, deliver 1, get 8, put 8 in pending, ask for 2,3,4,5,6,7 2 time out, | |
| | | Put 2(nil) in pending |
| | 1 time out | |
| | Delivers[s] = 1 | Delivers[s] = 1 Deliver 2(empty) |
| | Get 3,4,5,6,7 | |
| | Put 3,4,5,6,7 in pending | Deliver 3,4,5,6,7,8 |

From the example we can easily recognize that the Tutorial Algorithm is much better than Textbook.

Exercise 4. Experiment with the Lazy Probabilistic Broadcast

The loss rate of the links and parameters of the gossip, such as the fanout, threshold, and maxround influences the reliability of the broadcast.

If we have significant loss rate on communication links, then broadcast cannot work well. First of all, the initial broadcast made by one node may reach only few other nodes, even worse, later on the some nodes gossip to others, their requests may lost, as well as the responding message if any, therefore it becomes unreliable on very lossy links, however the gossip algorithm still is able to compensate faulty processes or links with omission failures by information dissemination, rather than collecting acknowledgements.

A higher value of fanout and maxround will increase probability of finding a lost message by consulting significant amount of nodes, however the higher the fanout and maxround, the higher the load imposed on each process and the amount of redundant information exchanged in the network.

The number of rounds needed for a message to be found depends on the fanout. Roughly speaking, the higher the fanout, the higher is the number of messages exchanged at every round, and consequently the lower is the number of rounds needed for a message to be found.

On the other hand, the higher the threshold, the less is the chance that certain node stores a copy of message, and then other nodes can find the message nowhere if no one keep it, therefore it's not reliability to have big threshold.

| | Large | Small |
|------------------|------------------|------------------|
| LossRate | Low Reliability | High Reliability |
| Fanout | High Reliability | Low Reliability |
| Threshold | Low Reliability | High Reliability |
| MaxRound | High Reliability | Low Reliability |

Description of three scenarios:

Never Lost

Description:

Do Probabilistic Broadcast from node 0 to all nodes.

Topology:

Set all link with loss_rate="0"

Node 0:

Application THREAD RUNNING

Enter Message to do ReliableBroadcast:

a

Start ProbabilisticBroadcast message: a

Raising ProbabilisticBroadcastEvent

Enter Message to do ReliableBroadcast:

||| store message a, init:ref://127.0.0.1:13340/0, seq:1

--- pbDeliverEvent, msg is a, init:ref://127.0.0.1:13340/0, seq:1, from:ref://127.0.0.1:13340/0

b

Start ProbabilisticBroadcast message: b

Raising ProbabilisticBroadcastEvent

Enter Message to do ReliableBroadcast:

||| store message b, init:ref://127.0.0.1:13340/0, seq:2

--- pbDeliverEvent, msg is b, init:ref://127.0.0.1:13340/0, seq:2, from:ref://127.0.0.1:13340/0

c

Start ProbabilisticBroadcast message: c

Raising ProbabilisticBroadcastEvent

Enter Message to do ReliableBroadcast:

--- pbDeliverEvent, msg is c, init:ref://127.0.0.1:13340/0, seq:3, from:ref://127.0.0.1:13340/0

Node 5:

Application THREAD RUNNING

Enter Message to do ReliableBroadcast:

--- pbDeliverEvent, msg is a, init:ref://127.0.0.1:13340/0, seq:1, from:ref://127.0.0.1:13340/0

--- pbDeliverEvent, msg is b, init:ref://127.0.0.1:13340/0, seq:2, from:ref://127.0.0.1:13340/0

--- pbDeliverEvent, msg is c, init:ref://127.0.0.1:13340/0, seq:3, from:ref://127.0.0.1:13340/0

Summary:

Node 0 broadcast message with no loss_rate link. So Node 5 gets these messages directly from node 0 and no message missing.

Lost and Found

Description:

Node 0 has a loss_rate = 0.5 link with Node 5. If node 5 found missing a message he will ask the message by gossip

Topology:

```
...
<link src_id="0" dst_id="5" latency="1000" loss_rate="0.5" undirected="true"/>
...
```

Node 0:

Application THREAD RUNNING

Enter Message to do ReliableBroadcast:

a

Start ProbabilisticBroadcast message: a

Raising ProbabilisticBroadcastEvent

Enter Message to do ReliableBroadcast:

{DropComponent} Message BroadcastMessage to 5 dropped

--- pbDeliverEvent, msg is a, init:ref://127.0.0.1:13340/0, seq:1, from:ref://127.0.0.1:13340/0

b

Start ProbabilisticBroadcast message: b

Raising ProbabilisticBroadcastEvent

Enter Message to do ReliableBroadcast:

{DropComponent} Link to 5 has loss rate of: 0.5 msg=assignment3.events.BroadcastMessage@29ab3e

Node 5

Application THREAD RUNNING

Enter Message to do ReliableBroadcast:

miss message, init:ref://127.0.0.1:13340/0, seq:1, start gossip

+++ Gossip +++

gossip to 4, maxRounds:2, seq:12, init:ref://127.0.0.1:13340/0

gossip to 0, maxRounds:2, seq:12, init:ref://127.0.0.1:13340/0

gossip to 3, maxRounds:2, seq:12, init:ref://127.0.0.1:13340/0

receive a pbBroadcastRequestMessage from ref://127.0.0.1:13344/4 init:ref://127.0.0.1:13340/0 seq:1, replyTo:ref://127.0.0.1:13345/5

%%% nothing found, start gossip

+++ Gossip +++

gossip to 4, maxRounds:0, seq:10, init:ref://127.0.0.1:13340/0

gossip to 1, maxRounds:0, seq:10, init:ref://127.0.0.1:13340/0

gossip to 3, maxRounds:0, seq:10, init:ref://127.0.0.1:13340/0

receive a pbBroadcastRequestMessage from ref://127.0.0.1:13343/3 init:ref://127.0.0.1:13340/0 seq:1,
replyTo:ref://127.0.0.1:13345/5

%%% nothing found, start gossip

+++ Gossip +++

gossip to 4, maxRounds:0, seq:10, init:ref://127.0.0.1:13340/0

gossip to 1, maxRounds:0, seq:10, init:ref://127.0.0.1:13340/0

gossip to 2, maxRounds:0, seq:10, init:ref://127.0.0.1:13340/0

receive a pbBroadcastRequestMessage from ref://127.0.0.1:13342/2 init:ref://127.0.0.1:13340/0 seq:1,
replyTo:ref://127.0.0.1:13345/5

%%% nothing found, maxRound = 0, no gossip

receive a pbBroadcastRequestMessage from ref://127.0.0.1:13341/1 init:ref://127.0.0.1:13340/0 seq:1,
replyTo:ref://127.0.0.1:13345/5

%%% nothing found, maxRound = 0, no gossip

timeOutEvent

--- pbDeliverEvent, msg is b, init:ref://127.0.0.1:13340/0, seq:2, from:ref://127.0.0.1:13340/0

Summary:

Node 5 lost message “a”, and ask it by gossip. And finally gets it from node 0

Lost Forever

Description:

Node 0 has a loss_rate = 0.5 link with Node 5. If node 5 found missing a message he will ask the message by gossip, but no one store the message. So the message will lose forever.

Topology:

```
...
<link src_id="0" dst_id="5" latency="1000" loss_rate="0.5" undirected="true"/>
...
```

Lpb.properties:

```
...
store.threshold = 1
...
```

Node 0:

Application THREAD RUNNING

Enter Message to do ReliableBroadcast:

a

Start ProbabilisticBroadcast message: a

Raising ProbabilisticBroadcastEvent

Enter Message to do ReliableBroadcast:

{DropComponent} Message BroadcastMessage to 5 dropped

--- pbDeliverEvent, msg is a, init:ref://127.0.0.1:13340/0, seq:1, from:ref://127.0.0.1:13340/0

b

Start ProbabilisticBroadcast message: b

Raising ProbabilisticBroadcastEvent

Enter Message to do ReliableBroadcast:

{DropComponent} Link to 5 has loss rate of: 0.5 msg=assignment3.events.BroadcastMessage@41d05d

--- pbDeliverEvent, msg is b, init:ref://127.0.0.1:13340/0, seq:2, from:ref://127.0.0.1:13340/0

Node 5:

Application THREAD RUNNING

Enter Message to do ReliableBroadcast:

miss message, init:ref://127.0.0.1:13340/0, seq:1, start gossip

+++ Gossip +++

gossip to 4, maxRounds:2, seq:12, init:ref://127.0.0.1:13340/0

gossip to 1, maxRounds:2, seq:12, init:ref://127.0.0.1:13340/0

gossip to 3, maxRounds:2, seq:12, init:ref://127.0.0.1:13340/0

receive a pbBroadcastRequestMessage from ref://127.0.0.1:13344/4 init:ref://127.0.0.1:13340/0 seq:1,

replyTo:ref://127.0.0.1:13345/5

%%% nothing found, start gossip

```
+++ Gossip +++
gossip to 4, maxRounds:0, seq:10, init:ref://127.0.0.1:13340/0
gossip to 0, maxRounds:0, seq:10, init:ref://127.0.0.1:13340/0
gossip to 2, maxRounds:0, seq:10, init:ref://127.0.0.1:13340/0
receive a pbBroadcastRequestMessage from ref://127.0.0.1:13343/3 init:ref://127.0.0.1:13340/0 seq:1,
replyTo:ref://127.0.0.1:13345/5
%%% nothing found, start gossip

+++ Gossip +++
gossip to 4, maxRounds:0, seq:10, init:ref://127.0.0.1:13340/0
gossip to 1, maxRounds:0, seq:10, init:ref://127.0.0.1:13340/0
gossip to 3, maxRounds:0, seq:10, init:ref://127.0.0.1:13340/0
receive a pbBroadcastRequestMessage from ref://127.0.0.1:13341/1 init:ref://127.0.0.1:13340/0 seq:1,
replyTo:ref://127.0.0.1:13345/5
%%% nothing found, start gossip

+++ Gossip +++
gossip to 4, maxRounds:0, seq:10, init:ref://127.0.0.1:13340/0
gossip to 2, maxRounds:0, seq:10, init:ref://127.0.0.1:13340/0
gossip to 3, maxRounds:0, seq:10, init:ref://127.0.0.1:13340/0
receive a pbBroadcastRequestMessage from ref://127.0.0.1:13344/4 init:ref://127.0.0.1:13340/0 seq:1,
replyTo:ref://127.0.0.1:13345/5
%%% nothing found, maxRound = 0, no gossip
timeOutEvent
--- pbDeliverEvent, msg is b, init:ref://127.0.0.1:13340/0, seq:2, from:ref://127.0.0.1:13340/0
```

Summary:

Node 5 lost message “a”, and ask it by gossip. Since no one store message and it cannot found message “a”