|  |
| --- |
| Software Engineering of Distributed Systems, KTH |
| Distributed Systems Advanced Homework 3 |
| Implementation of Reliable Broadcast Component, Unreliable Broadcast Component and Lazy Probabilistic Broadcast Component |

|  |
| --- |
| Shanbo Li and Sike Huang  2/19/2008 |

### Exercise 1. Modify Algorithm 3 such that it garbage collects the delivered set. Messages that no longer need to be maintained in the delivered set should be removed. Update your implementation of RB and describe the new algorithm in the report.

**Answer:**

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} | |
| 1. p0 sends m to all | |
| 1. p0 crashes and failed to send m to p1 | |
| 1. p2, p3, p4 get m | |
| 1. p2 detects p0 crashed and broadcast m to all nodes. | |
| 1. p1 gets m for the first time from p2 | (counter(m) = 1, correct.size =4 ) |
| 1. p3, p4 detect p0 crash and broadcast m to all nodes. | |
| 1. p1 gets m from p3, p4 | (counter(m) = 3, correct.size = 4) |
| 1. p3, p4 crash. | |
| 1. p2 crashes | |
| 1. p1 detects p2 crashed and broadcast m to all nodes. | |
| 1. p1 gets m from himself. | (counter(m) = 4, correct.size = 1) |
| 1. 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)>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 3. The Lazy Probabilistic Broadcast (Algorithms 5, 6 and 7) presented here is different from Algorithm 3.10-3.11 from the textbook (pages 92-93). Analyze the differences between the two algorithms and discuss their implications in the written report.

**Answer:**

**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.

|  |
| --- |
| No message 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 linke. So Node 5 gets these message directly from node 0 and no message missing. |

|  |
| --- |
| a message is lost by the unreliable broadcast but  recovered by gossip, |
| 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 linke. So Node 5 gets these message directly from node 0 and no message missing. |