Exercise Session 7
GM and VSC

November 9, 2015

Problem 1

Show that P is the weakest failure detector for Group Membership.

Solution

In order to show that P is the weakest failure detector for Group Membership, we need to show that:

- P can be used to implement Group Membership.
- Group Membership can be used to implement P.

The first direction stems directly from the Group Membership implementation in the class.

For the second direction, we assume that all processes run Group Membership algorithm. Whenever a new view is installed, all processes that are freshly removed from the view are added to the suspected set. This approach satisfies both *Strong Completeness* and *Strong Accuracy* of P, directly from the corresponding properties of Group Membership.

Problem 2

Note: This is a continuation of Problem 3 from Exercise Session 6 (last week).

In this problem we will change the *view-synchronous communication (VSC)* abstraction in order to allow joins of new processes. Answer to the following questions:

1. ~~Are the properties of VSC (as given in the class) suitable to accommodate the joins of new processes. Why / Why not?~~ (done)
2. ~~Change the properties of VSC, so that they allow for implementations that support the joins of new processes. (Hint: focus on the properties of *group membership*)~~ (done)
3. Sketch the changes we need to perform on the Consensus-based (Algorithm II) implementation of VSC in order to support joins.

Solution 2.3

The solution is described in Algorithm 1, 2 on the last two pages of this document. The changes to the regular algorithm are highlighted in red (note that we used the consensus algorithm that appears in the book — it is similar in spirit to the version in the slides).

We add two new local variables to the algorithm: $joined$ and $crashed$. The $joined$ variable is a boolean flag that is set to true after the process successfully joins a view (is part of the view members). The $joined$ flag differentiates the behavior of processes that are just attempting to join. The $crashed$ variable is a local set that keeps track of crash events received from the failure detector. This set is useful in executions where a process p attempts to join and then crashes. If another correct process $p2$ sees the join attempt only after the crash notification, it needs to remember that it has already seen a crash of p and to disregard the join.

For most events, the only difference to the original algorithm is that we impose the condition $joined = true$ for event handlers. Recall that such a conditional event handler means that the events are implicitly buffered until the condition becomes true (see the document describing the language used for module specification in "Additional Material" section on the course website). For example, the crash handler is now conditioned by $joined = true$. This means that any crash event received by the process while it is still joining will be buffered. The events will, however, be handled right after the process successfully joins a view.

The joining begins when the application emits a $Join$ event (line 21). If the process has not joined yet and is not part of the initial set of processes in the view, the process broadcasts a $JoinReq$ message to every other process. The $JoinReq$ message can be seen as a dual of the $crash$ event. It will be the job of the receiving correct processes (that are already view members) to handle the join and propose the addition of the joining process to a view.

Upon receiving a $JoinReq$ message (line 23), processes will add the joining process to their correct set. Note that if the receiving process has already seen a crash of the joining process, the correct set will not be changed ($\{p\} \setminus crashed$ will be $\emptyset$). Changing the $correct$ set will trigger the handler at line 37 and initiate a view change. Processes that have seen the broadcast from the joining process will propose it in the new view member set. Since the joining process uses best-effort broadcast, correct processes will eventually receive the $JoinReq$ broadcast message (if the joining process is also correct).

Another difference with the initial algorithm is that once a decision is taken in the consensus and a process moves to a new view, every process broadcasts the new view (both its member set and id). This broadcast is useful for joining processes. If a joining process sees that it is part of a new view, it will initialize its view id, member set and correct set accordingly. Finally, the joining process sets the $joined$ flag to true (meaning that it will handle all buffered events) and emits a $JoinOk$ indication to the application.

Algorithm 1 View synchrony with joins, first part

```
1: Implements:
2:     VSCJ (vscj)
3: Uses:
4:     UniformConsensus (ucons)
5:     BestEffortBroadcast (beb)
6:     PerfectFailureDetector (P)

7: upon event ⟨vscj, Init⟩ do
8:     (vid, M) := (0, Π)
9:     correct := Π
10:    flushing := false; blocked := false; wait := false;
11:    pending := ∅; delivered := ∅; crashed := ∅
12:    forall m do ack[m] := ∅
13:    seen := [⊥]^N
14:    trigger ⟨vscj, View | (vid, M)⟩
15:    if self ∈ Π then
16:        joined := true
17:    else
18:        joined := false
19:    end if

20:    upon event ⟨vscj, Broadcast | m⟩ such that blocked = false∧joined = true do
21:        pending := pending ∪ (self, m)
22:        trigger ⟨beb, Broadcast | [DATA, vid, self, m]⟩

23:    upon event ⟨vscj, Deliver | p, [DATA, id, s, m]⟩ such that joined = true do
24:        if id = vid ∧ blocked = false then
25:            ack[m] := ack[m] ∪ {p}
26:            if (s, m) ∉ pending then
27:                pending := pending ∪ (s, m)
28:                trigger ⟨beb, Broadcast | [DATA, vid, s, m]⟩
29:            end if
30:        end if

31:    upon ∃(s, m) ∈ pending : M ⊆ ack[m] ∧ m ∉ delivered∧joined = true do
32:        delivered := delivered ∪ {m}
33:        trigger ⟨vscj, Deliver | s, m⟩

34:    upon event ⟨P, Crash | p⟩ such that joined = true do
35:        correct := correct \ {p}
36:        crashed := crashed ∪ {p}

37:    upon correct ≠ M ∧ flushing = false∧joined = true do
38:        flushing := true
39:        trigger ⟨vscj, Block⟩

40:    upon event ⟨vscj, BlockOk⟩ such that joined = true do
41:        blocked := true
42:        trigger ⟨beb, Broadcast | [PENDING, vid, pending]⟩

43:    upon event ⟨beb, Deliver | p, [PENDING, id, pd]⟩ such that id = vid∧joined = true do
44:        seen[p] := pd

45:    upon ∀p ∈ correct : seen[p] ≠ ⊥ ∧ wait = false do
46:        wait := true
47:        vid := vid + 1
48:        initialize a new instance uc.vid of uniform consensus
49:        trigger ⟨uc.vid, Propose | (correct, seen)⟩
```

Algorithm 2 View synchrony with joins, second part

```
1:  upon event ⟨uc.id, Decide | M', S⟩ do
2:      ∀p ∈ M' : S[p] ≠ ⊥ do
3:          ∀(s, m) ∈ S[p] : m ∉ delivered do
4:              delivered := delivered ∪ {m}
5:              trigger ⟨vscj, Deliver | s, m⟩
6:      flushing := false; blocked := false; wait := false
7:      pending = ∅
8:      ∀m do ack[m] := ∅
9:      seen := [⊥]^N
10:     M := M'
11:     trigger ⟨vscj, View | (vid, M)⟩
12:     ∀p ∈ M do
13:         trigger ⟨beb, Broadcast | [NewView, vid, M]⟩

14:     upon event ⟨beb, Deliver | [NewView, vid', M']⟩ such that joined = false do
15:         if self ∈ M' then
16:             (vid, M) := (vid', M')
17:             correct := M
18:             joined := true
19:             trigger ⟨vscj, JoinOk⟩
20:         end if

21:     upon event ⟨vscj, Join | self⟩ such that joined = false do
22:         trigger ⟨beb, Broadcast | [JoinReq, self]⟩

23:     upon event ⟨beb, Deliver | [JoinReq, p]⟩ such that joined = true do
24:         correct := correct ∪ {p} \ crashed
```