Distributed Algorithms

Fall 2019

GM & VSC

8th exercise session, 18/11/2019

Matteo Monti <matteo.monti@epfl.ch>

Athanasios Xygkis <athanasios.xygkis@epfl.ch>

Exercise 1 - Weakest failure detector

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

Note: the failure detector *D* is weakest for solving some problem A (e.g., Consensus or NBAC) if D provides the smallest amount of information about failures that allows to solve A.

Hint: Reduce Group Membership to Perfect Failure Detector and vice versa.

Exercise 1 - Solution (1/2)

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

1. P can be used to implement Group Membership.
2. Group Membership can be used to implement P.

For (1), the proof is the algorithm given in the class.

- The perfect FD satisfies the completeness and accuracy properties of GM,
- The Uniform Consensus (an implementation of which also uses the perfect FD) satisfies agreement property of GM,
- The fact that a process forms a new view only when the set of correct processes is properly contained in the current view membership satisfies the new monotonicity property.

Exercise 1 - Solution (2/2)

For (2), assume that all processes run a GM algorithm. We can implement a perfect failure detector as follows:

Whenever a new view is installed, all processes that are freshly removed from the view are added to the detected set.

This approach satisfies both Strong Completeness and Strong Accuracy, directly from the corresponding properties of GM.

Exercise 2 - Properties of VSC with Joins (1/2)

The view-synchronous communication (VSC) abstraction presented in class does not allow joins of new processes.

Answer to the following questions:

1. Why are the properties of VSC (as given in the class) not suitable for accommodating the joins of new processes?

Exercise 2 - Solution of part 1 (1/2)

The properties of VSC for the case of crashes:

1. Properties of Reliable Broadcast (Validity, No creation, No duplication, Agreement),
2. Properties of Group Membership,
3. Extra property (View inclusion): If some process delivers a message m from process p in view V, then m was broadcast by p in view V.

Exercise 2 - Solution of part 1 (2/2)

The properties of GM - as presented in the class - are crash-specific:

1. Local monotonicity: If a process p install a view V = (i, M) and subsequently installs a view V’=(j, M’), then i < j and M⊇M’,
   - **The fact the view always shrinks is true for crashes, but not for joins**
2. Uniform Agreement: If some process installs a view V=(i, M), and another process installs some view V’=(i, M’), then M=M’,
   - **This one is OK**
3. Completeness: If a process p crashes, then eventually every correct process install a view (i, M), such that p∉M,
   - **Assumes crashes, does not impose any condition for joins**
4. Accuracy: If some process installs a view (i, M) with q∉M for some process q∈Π, then q has crashed
   - **Assumes crashes, does not impose any condition for joins**

Conclusion: Properties 1, 3, 4 are not suitable for Joins and need to be rectified.

Exercise 2 - Properties of VSC with Joins (2/2)

2. Change the properties of VSC, so that they allow for implementations that support the joins of new processes.

   Assume that these implementations already provide the events <*Join*|*p*> and <*JoinOK*> and focus solely on the properties.

Hint: focus on the properties of group membership.

Exercise 2 - Solution of part 2 (1/5)

Local Monotonicity. There are two ways to rectify monotonicity for joins:

1. LM1 (Not not allow no-op view changes): If a process installs a view V=(i, M) and then installs view (i+1, N), then M≠N.

With LM1, notice that:

a. Consecutive views must different set of processes.

b. Different views can have the set of processes. E.g. if process q joins and then crashes the sequence of view will be: (i, M) → (i+1, M ∪ {q}) → (i+2, M).

c. Processes can repeatedly be included and excluded from a view (i.e. no monotonicity ~ view oscillation). E.g. process q joins, crashes, restarts, joins again.

Exercise 2 - Solution of part 2 (2/5)

2. LM2 (Do not allow view oscillations, i.e. definite crashes): If a process p installs views (i, N) and (j, M) where j>i, q∈N, and q∉M, then for all k>j, if p installs view (k, O), then q∉O.

With LM2, notice that:

a. Once a process is excluded from a view it can never come back.
b. In practice, for a process to come back a new process id must be adopted.

Exercise 2 - Solution of part 2 (3/5)

Completeness:

1. C1 (If we adopt LM1):
   a. If a process p crashes, then ∃ i∈ℕ such that for all correct processes q, if j>i and q install view (j,M), then p∉M.

      This statement says that a crashed process is removed from a future view, not from all future views!

   b. If a correct process q requests to join,then ∃ i∈ℕ such that every correct process installs view (i, M), such that p∈M.

      This statement that processes that want to join, eventually join!

2. C2 (If we adopt LM2):
   a. Stays the same, i.e. if a process p crashes, then eventually every correct process install a view (i, M), such that p∉M.
   b. Same as (b) from C1.

Exercise 2 - Solution of part 2 (4/5)

The following properties apply to both LM1/C1 and LM2/C2.

Accuracy:

If a process p installs views (i, M) and (i+1, N) where q∈M but q∉N, then q has crashed.

Notice the difference with the accuracy property given in the class. This property detects crashes by comparing views, since new processes can join during the execution (i.e. the set Π is not known in advance).

Validity:

To be technically correct, we require the following property:

If some process p installs a view (i, M) and some process q is in M, then q previously requested to join or q∈Π.

Exercise 2 - Solution of part 2 (5/5)

The following properties apply to both LM1/C1 and LM2/C2.

The agreement property of the Reliable Broadcast requires changing.

1. A1 (Keep original agreement property):
   a. If a message m is delivered by some correct process, then m is eventually delivered by every correct process.
   b. For (a) to hold, we have to make sure that when a process joins, it has to catch-up of all previously delivered messages.
2. A2 (Deliver only messages from views that the process participates in):
   a. If a message m is delivered by some correct process in view (i, M), then m is eventually delivered by all the processes belonging to M.
   b. This way, if $p \notin M$, then p does not have to deliver m.

View Inclusion: We keep it unchanged.

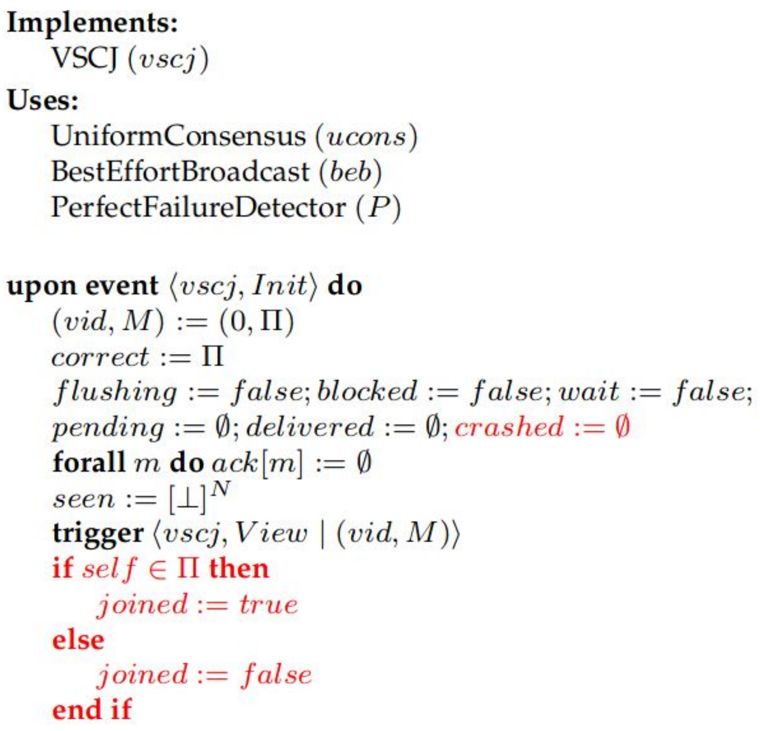
Note: The view inclusion does not conflict with the agreement property above. The view inclusion talk about “where to deliver”, while the agreement property talk about “when to deliver”.

Exercise 3 - VSC with Joins (Bonus)

Implement joins on the Consensus-based algorithm (Algorithm II) of VSC.

Exercise 3 - Solution (1/5)

Modifications are shown in red



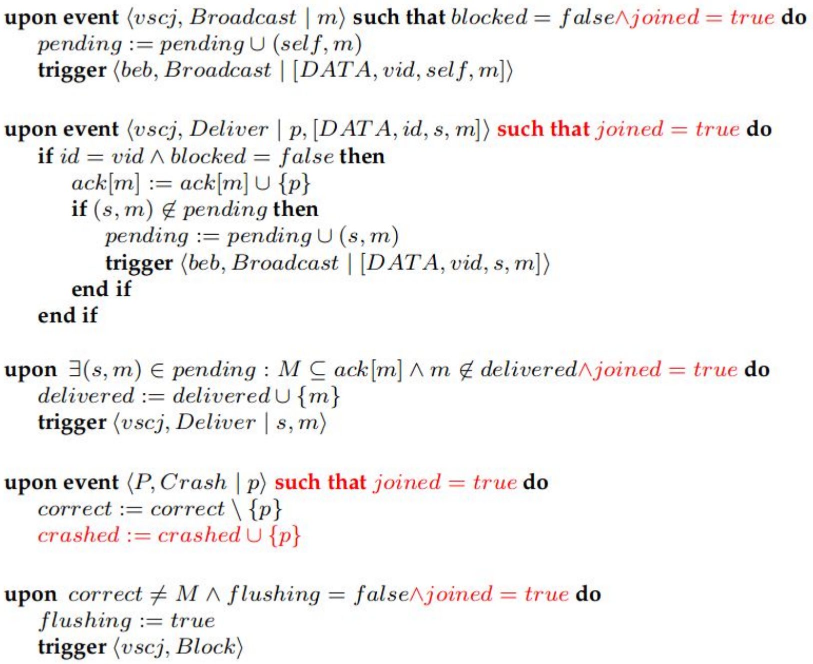
```
Implements:
    VSCJ ($vscj$)
Uses:
    UniformConsensus ($ucons$)
    BestEffortBroadcast ($beb$)
    PerfectFailureDetector ($P$)

upon event $\langle vscj, Init \rangle$ do
    $(vid, M) := (0, \Pi)$
    $correct := \Pi$
    $flushing := false; blocked := false; wait := false;$
    $pending := \emptyset; delivered := \emptyset; crashed := \emptyset$
    forall $m$ do $ack[m] := \emptyset$
    $seen := [\bot]^N$
    trigger $\langle vscj, View \mid (vid, M) \rangle$
    if $self \in \Pi$ then
        $joined := true$
    else
        $joined := false$
    end if
```

- *crashed* is a set that tracks the events from the failure detector.
    - It is useful in executions where a process attempts to join and then crashes. If another correct process “sees” the join attempt *after* the crash notification, it uses the *crashed* set to disregard the join.
- *joined* is set to true after the process successfully joins a view.

Exercise 3 - Solution (2/5)

Modifications are shown in red



```
upon event $\langle vscj, Broadcast \mid m \rangle$ such that $blocked = false \wedge joined = true$ do
    $pending := pending \cup (self, m)$
    trigger $\langle beb, Broadcast \mid [DATA, vid, self, m] \rangle$

upon event $\langle vscj, Deliver \mid p, [DATA, id, s, m] \rangle$ such that $joined = true$ do
    if $id = vid \wedge blocked = false$ then
        $ack[m] := ack[m] \cup \{p\}$
        if $(s, m) \notin pending$ then
            $pending := pending \cup (s, m)$
            trigger $\langle beb, Broadcast \mid [DATA, vid, s, m] \rangle$
        end if
    end if

upon $\exists (s, m) \in pending : M \subseteq ack[m] \wedge m \notin delivered \wedge joined = true$ do
    $delivered := delivered \cup \{m\}$
    trigger $\langle vscj, Deliver \mid s, m \rangle$

upon event $\langle P, Crash \mid p \rangle$ such that $joined = true$ do
    $correct := correct \setminus \{p\}$
    $crashed := crashed \cup \{p\}$

upon $correct \neq M \wedge flushing = false \wedge joined = true$ do
    $flushing := true$
    trigger $\langle vscj, Block \rangle$
```

This is the original algorithm, except that we have the extra condition:

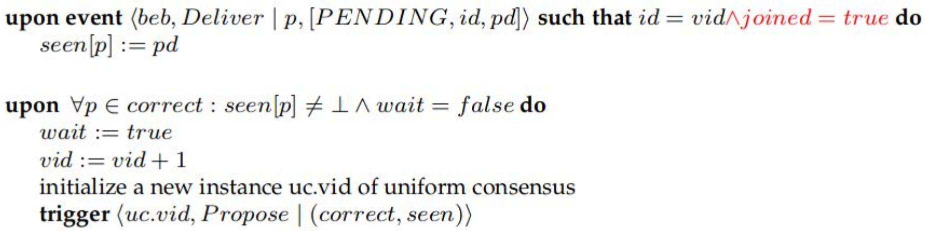
The logic should execute only for processes that have successfully joined a view (e.g. the initial set of processes).

Note: correct and crashed set are update together.

Also, the condition changed to *correct* $\neq$*M*, instead of *correct*$\subsetneq$*M*

Exercise 3 - Solution (3/5)

Modifications are shown in red



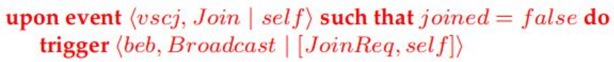
```
upon event $\langle beb, Deliver \mid p, [PENDING, id, pd]\rangle$ such that $id = vid \wedge joined = true$ do
    $seen[p] := pd$

upon $\forall p \in correct : seen[p] \neq \perp \wedge wait = false$ do
    $wait := true$
    $vid := vid + 1$
    initialize a new instance uc.vid of uniform consensus
    trigger $\langle uc.vid, Propose \mid (correct, seen)\rangle$
```

No surprises here

Exercise 3 - Solution (4/5)

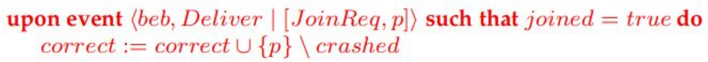
Modifications are shown in red



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

The joining begins when a process emits a join event. If a process has not joined yet and is not part of the initial set of processes in the view, the process broadcasts a *JoinReq* message.

Since the joining process uses BEB-broadcast, correct processes will eventually receive the JoinReq broadcast (if, of course, the joining process is also correct).



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

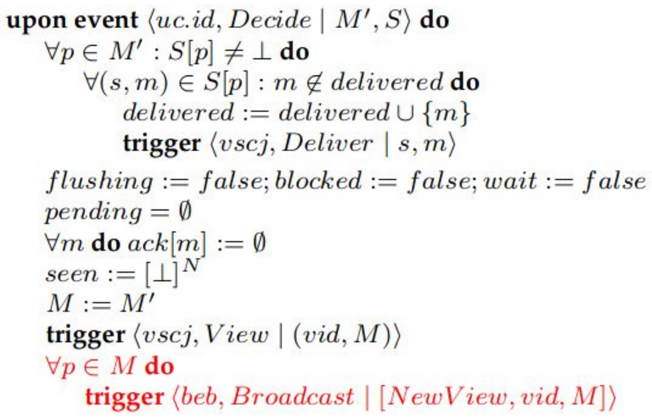
Upon receiving a *JoinReq*, processes will add the joining processes to their correct set. Changing the correct set will trigger the handler "**upon correct ≠M …**" and initiate a view change.

Note:

- It is the job of the receiving correct processes (that are already view members) to handle the *JoinReq* and propose the addition of the joining process to a view.
- If the receiving process has already seen a crash of the joining process, the correct set will not be changed, since {p}\crashed will be ∅.

Exercise 3 - Solution (5/5)

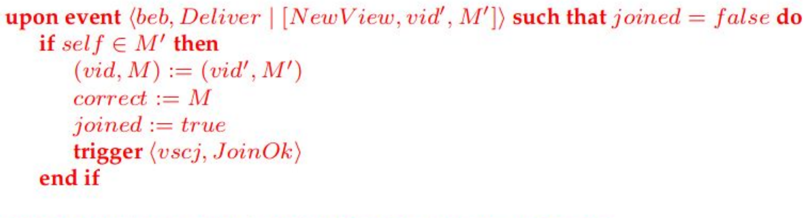
Modifications are shown in red



```
upon event ⟨$uc.id$, $Decide$ | $M', S$⟩ do
    $\forall p \in M' : S[p] \neq \bot$ do
        $\forall (s, m) \in S[p] : m \notin delivered$ do
            $delivered := delivered \cup \{m\}$
            trigger ⟨$vscj$, $Deliver$ | $s, m$⟩
    $flushing := false; blocked := false; wait := false$
    $pending = \emptyset$
    $\forall m$ do $ack[m] := \emptyset$
    $seen := [\bot]^N$
    $M := M'$
    trigger ⟨$vscj$, $View$ | $(vid, M)$⟩
    $\forall p \in M$ do
        trigger ⟨$beb$, $Broadcast$ | $[NewView, vid, M]$⟩
```

Processes that have seen the broadcast from the joining process will propose it in the new view member set.

Also, every process broadcasts the new view (both its member set and id).



```
upon event ⟨$beb$, $Deliver$ | $[NewView, vid', M']$⟩ such that $joined = false$ do
    if $self \in M'$ then
        $(vid, M) := (vid', M')$
        $correct := M$
        $joined := true$
        trigger ⟨$vscj$, $JoinOk$⟩
    end if
```

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.