Exercise Session 8
View Synchronous Communication

Problem 1

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? (Hint: no)
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*)
3. Sketch the changes we need to perform on the Consensus-based (Algorithm II) implementation of VSC in order to support joins.

Solution

Solution 1.1

No, the properties are not suitable for joins. The most obvious property is Local Monotonicity. Joins imply that the set of correct processes in a view can increase, and this would break the local monotonicity property. Furthermore, Completeness and Accuracy only refer to crashes, without imposing any conditions on the correctness of joins.

Solution 1.2

First, we need to add a $\langle Join|p\rangle$ event to allow new processes to join the group. After a process emits such an event, we says that it requested to join. The VSC layer emits a $\langle JoinOk\rangle$ event to the application when it has successfully joined a view. The application can start emitting broadcast requests after it receives the $JoinOk$ event.

Group membership properties

Let us first look at the four group membership properties.

View Monotonicity. The monotonicity property of VSC (GM1) ensures that the number of processes in a view decreases over time. Since new processes can join, this need change: Three possibilities can be considered:

- Get rid of it entirely.
- Require that views do not change for nothing: If a process installs views (j, N) and $(j+1, M)$, then $M \neq N$.
- Require that views do not oscillate: if a process p installs views (i, M) and (j, N) where $j > i$, $q \in M$, and $q \notin N$, then for all $k > j$, if p installs (j, O), then $p \notin O$.

With the second option, the new property ensures that consecutive views have different sets of processes, i.e., that the view cannot change if there is no change in the correct set of processes. Notice, however, that it is still possible for two views to have the same set of processes, e.g., if a processes joins and then crashes. It is also possible for a process to repeatedly be included and excluded from a view.

With the third option, once a process is excluded from a view it can never come back.

Uniform agreement. The uniform agreement property of VSC (GM2) ensures that all processes install the same sequence of view. We will keep this property.

Completeness. If we choose the third version of monotonicity, then we can keep the completeness property of the group membership abstraction. If we choose one of the first two, we need to make some changes: Because the sequence of views is no longer monotonic, we need to strengthen a bit the completeness property of VSC (GM3): If a process p crashes, then there is $i \in \mathbb{N}$ such that for all correct process q, if $j > i$ and q installs view (j, M), then $p \notin M$.

To ensure that processes which want to join eventually join a view, we add the following completeness property: If a correct process p requests to join, then there is an integer i such that every correct process eventually installs view (i, M) such that $p \in M$.

Accuracy. If a process p installs views (i, M) and $(i + 1, N)$ where $q \in M$ but $q \notin N$, then q has crashed.

On top of those properties, we will also require that a process is included in a view only if it requested so.

Validity. If some process installs a view (i, M) and some process q is in M, then q previously requested to join or $q \in \Pi$.

Broadcast properties

Let us now look at the broadcast properties of VSC. Those are the same of for reliable broadcast (RB1,2,3,4). We need to decide whether a process which joins need to "catch-up" on all previously delivered messages or can just start with the messages of the first view in which it is included.

If we choose the first then we need to relax Agreement (RB4) so that a process need to deliver only the messages sent in the view to which it participates: If message m is delivered by some correct process in view (i, M), then m is eventually delivered by all the process belonging to M. This way, if $p \notin M$ then p does not have to deliver m.

If we choose the first solution then we can leave RB1,2,3,4 unchanged.

View Inclusion

Finally, we will keep the View Inclusion property as is.

Solution 1.3

The solution is described in Algorithm 1, 2. 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 Additional Material on the 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 $\langle vscj, Init\rangle$ do
8:     $(vid, M) := (0, \Pi)$
9:     $correct := \Pi$
10:    $flushing := false; blocked := false; wait := false;$
11:    $pending := \emptyset; delivered := \emptyset; crashed := \emptyset$
12:    forall $m$ do $ack[m] := \emptyset$
13:    $seen := [\perp]^N$
14:    trigger $\langle vscj, View \mid (vid, M)\rangle$
15:    if $self \in \Pi$ then
16:        $joined := true$
17:    else
18:        $joined := false$
19:    end if

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

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

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

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

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

40:    upon event $\langle vscj, BlockOk\rangle$ such that $joined = true$ do
41:        $blocked := true$
42:        trigger $\langle beb, Broadcast \mid [PENDING, vid, pending]\rangle$

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

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

Algorithm 2 View synchrony with joins, second part

```
1: upon event $\langle uc.id, Decide \mid M', S\rangle$ do
2:     $\forall p \in M' : S[p] \neq \bot$ do
3:         $\forall (s, m) \in S[p] : m \notin delivered$ do
4:             $delivered := delivered \cup \{m\}$
5:             trigger $\langle vscj, Deliver \mid s, m\rangle$
6:     $flushing := false; blocked := false; wait := false$
7:     $pending = \emptyset$
8:     $\forall m$ do $ack[m] := \emptyset$
9:     $seen := [\bot]^N$
10:    $M := M'$
11:    trigger $\langle vscj, View \mid (vid, M)\rangle$
12:    $\forall p \in M$ do
13:        trigger $\langle beb, Broadcast \mid [NewView, vid, M]\rangle$

14:    upon event $\langle beb, Deliver \mid [NewView, vid', M']\rangle$ such that $joined = false$ do
15:        if $self \in M'$ then
16:            $(vid, M) := (vid', M')$
17:            $correct := M$
18:            $joined := true$
19:            trigger $\langle vscj, JoinOk\rangle$
20:        end if

21:    upon event $\langle vscj, Join \mid self\rangle$ such that $joined = false$ do
22:        trigger $\langle beb, Broadcast \mid [JoinReq, self]\rangle$

23:    upon event $\langle beb, Deliver \mid [JoinReq, p]\rangle$ such that $joined = true$ do
24:        $correct := correct \cup \{p\} \setminus crashed$
```