CS 247: Principles of Distributed Computing
Time: 80 mins

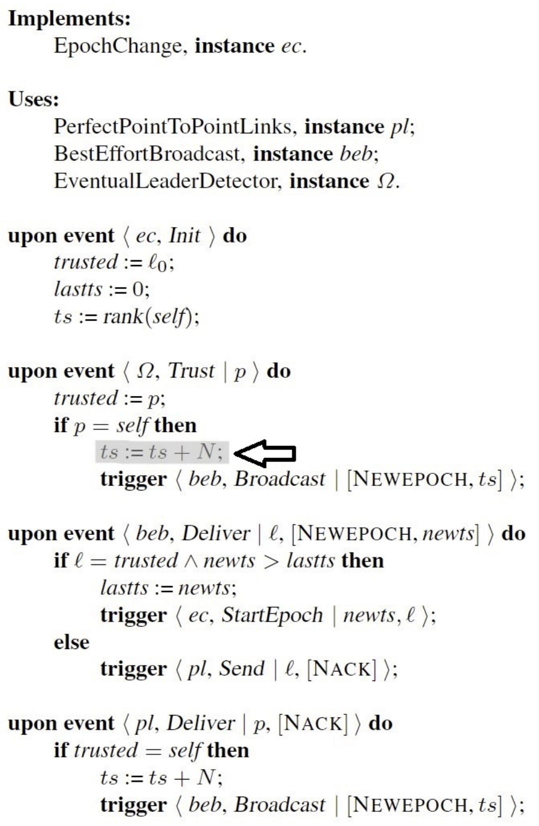
Name and ID: ____________________

Instructor's name: ____________________

1. (20 points) **Explain** whether the following statements are True or False.
   (a) The eventual leader detector can be implemented in a partially synchronous system.
   Answer:
   True
   (b) The total-order broadcast abstraction can be implemented in an asynchronous system where processes can crash.
   Answer:
   False due to FLP result
   (c) The total-order broadcast abstraction preserves causal ordering as well.
   Answer:
   False, tob only guarantees same order across nodes
   (d) The total-order broadcast abstraction is computationally more powerful that the consensus in systems with reliable channels.
   Answer:
   False

2. (20 points) Consider the following leader-based epoch change protocol with the following properties:

   - *Monotonicity*
     If a correct process starts an epoch (ts, l) and later starts an epoch (ts', l'), then ts' > ts.
   - *Consistency*
     If a correct process starts an epoch (ts, l) and another correct process starts an epoch (ts', l') with ts = ts', then l = l'.
   - *Eventual leadership*
     There is a time after which every correct process has started some epoch and starts no further epoch, such that the last epoch started at correct processes is epoch (ts, l) and process l is correct.



```
Implements:
    EpochChange, instance ec.

Uses:
    PerfectPointToPointLinks, instance pl;
    BestEffortBroadcast, instance beb;
    EventualLeaderDetector, instance Ω.

upon event ⟨ ec, Init ⟩ do
    trusted := $\ell_0$;
    lastts := 0;
    ts := rank(self);

upon event ⟨ Ω, Trust | p ⟩ do
    trusted := p;
    if p = self then
        ts := ts + N;   ⇦
        trigger ⟨ beb, Broadcast | [NEWEPOCH, ts] ⟩;

upon event ⟨ beb, Deliver | ℓ, [NEWEPOCH, newts] ⟩ do
    if ℓ = trusted ∧ newts > lastts then
        lastts := newts;
        trigger ⟨ ec, StartEpoch | newts, ℓ ⟩;
    else
        trigger ⟨ pl, Send | ℓ, [NACK] ⟩;

upon event ⟨ pl, Deliver | p, [NACK] ⟩ do
    if trusted = self then
        ts := ts + N;
        trigger ⟨ beb, Broadcast | [NEWEPOCH, ts] ⟩;
```

Figure 1: Epoch change protocol

Explain why the timestamp for each epoch (ts) is incremented by N (marked code location) and what happens if we replace that line with $ts := ts + 1$?
Answer:
To maintain consistency property timestamps at different nodes need to be partitioned. If it is incremented, the consistency property can be violated.

3. (20 points) Consider the uniform consensus III covered in the class. What happens if the eventual perfect failure detector is replaced with an unreliable failure detector which can output anything? Discuss the safety and liveness properties.
Answer:
Safety is never violated, but liveness can be violated. (5 points each) Need to explain each separately (5 points each)

4. (20 points) Use the indistinguishability argument to show that a Terminating Reliable Broadcast (TRB) abstraction cannot be implemented with an eventually perfect failure-detector ($\diamond$P) even if only one process can crash.
   Answer:
   Consider an execution E1, in which process s crashes initially and observe the possible actions for some correct process p: due to the termination property of TRB, there must be a time T at which p trb-delivers δ. Consider a second execution E2 that is similar to E1 up to time T, except that the sender s is correct and trb-broadcasts some message m, but all communication messages to and from s are delayed until after time T. The failure detector behaves in E2 as in E1 until after time T. This is possible because the failure detector is only eventually perfect. Up to time T, process p cannot distinguish E1 from E2 and trb-delivers δ. According to the agreement property of TRB, process s must trb-deliver δ as well, and s delivers exactly one message due to the termination property. But this contradicts the validity property of TRB, since s is correct, has trb-broadcast some message m = δ, and must trb-deliver m. (It is ok to draw executions instead

5. (20 points) Consider the view synchronous communication interface and protocol below:

Module:

Name: UniformViewSynchronousCommunication, **instance** *uvs*.

Events:

Request: $\langle$ *uvs*, *Broadcast* $\mid m \rangle$: Broadcasts a message m to all processes.

Indication: $\langle$ *uvs*, *Deliver* $\mid p, m \rangle$: Delivers a message m broadcast by process p.

Indication: $\langle$ *uvs*, *View* $\mid V \rangle$: Installs a new view $V = (id, M)$ with view identifier id and membership M.

Indication: $\langle$ *uvs*, *Block* $\rangle$
Request: $\langle$ *uvs*, *BlockOk* $\rangle$

Algorithm : TRB-Based View-Synchronous Communication

```
Implements:
    ViewSynchronousCommunication, instance vs.

Uses:
    UniformTerminatingReliableBroadcast (multiple instances);
    GroupMembership, instance gm;
    BestEffortBroadcast, instance beb.

upon event ⟨ vs, Init ⟩ do
    $(vid, M) := (0, \emptyset)$;
    flushing := FALSE; blocked := TRUE;
    inview := $\emptyset$;
    delivered := $\emptyset$;
    pendingviews := [];
    trbdone := $\emptyset$;

upon event ⟨ vs, Broadcast | $m$ ⟩ such that blocked = FALSE do
    inview := inview $\cup \{(self, m)\}$;
    delivered := delivered $\cup \{m\}$;   ⇦
    trigger ⟨ vs, Deliver | self, $m$ ⟩;
    trigger ⟨ beb, Broadcast | [DATA, $vid$, $m$] ⟩;

upon event ⟨ beb, Deliver | $p$, [DATA, $id$, $m$] ⟩ do
    if $id = vid \wedge$ blocked = FALSE $\wedge\ m \notin$ delivered then
        inview := inview $\cup \{(p, m)\}$;
        delivered := delivered $\cup \{m\}$;
        trigger ⟨ vs, Deliver | $p$, $m$ ⟩;

upon event ⟨ gm, View | $V'$ ⟩ do
    if $V' = (0, M')$ for some $M'$ then
        $(vid, M) := (0, M')$;
        blocked := FALSE;
    else
        append(pendingviews, $V'$);

upon pendingviews $\neq$ [] $\wedge$ flushing = FALSE do
    flushing := TRUE;
    trigger ⟨ vs, Block ⟩;

upon event ⟨ vs, BlockOk ⟩ do
    blocked := TRUE;
    forall $p \in M$ do
        Initialize a new instance utrb.$vid$.$p$ of uniform
        terminating reliable broadcast with sender $p$;
        if $p$ = self then
            trigger ⟨ utrb.$vid$.$p$, Broadcast | inview ⟩;

upon event ⟨ utrb.$id$.$p$, Deliver | $p$, $iv$ ⟩ such that $id = vid$ do
    trbdone := trbdone $\cup \{p\}$;
    if $iv \neq \triangle$ then
        forall $(s, m) \in iv$ such that $m \notin$ delivered do
            delivered := delivered $\cup \{m\}$;
            trigger ⟨ vs, Deliver | $s$, $m$ ⟩;

upon trbdone $= M \wedge$ blocked = TRUE do
    $V$ := head(pendingviews); remove(pendingviews, $V$);
    $(vid, M) := V$;
    correct := correct $\cap M$;
    flushing := FALSE; blocked := FALSE;
    inview := $\emptyset$;
    trbdone := $\emptyset$;
    trigger ⟨ vs, View | $(vid, M)$ ⟩;
```

Figure 2: View Synchronous Communication Interface and Protocol

(a) Explain the purpose of *Block* request and *BlockOK* indication.
Answer:
In the sildes.

(b) When broadcasting a message using this abstraction, can we postpone the delivery of the message to the current process (self)? In other words, can we remove the lines marked with an arrow in the *Broadcast* event? If yes, explain why. If no, describe an execution where this causes problem.
Answer:
It can violate the validity property if a view change is started right after brodcasting the message to others. A message that is broadcasted by p is not delivered by p itself (any example that indicates this is correct. Also, agreement can be violated and is an acceptable answer)