NBAC (Problem 2)

The problem

- Give an algorithm which implements NBAC in an asynchronous environment, using a *Reliable broadcast*, *Best-effort broadcast (beb)*, and failure detectors ?P, and <>S.

What is NBAC?

- NBAC1. Agreement: No two processes decide differently
- NBAC2. Termination: Every correct process eventually decides
- NBAC3. Commit-Validity: 1 can only be decided if all processes propose 1
- NBAC4. Abort-Validity: 0 can only be decided if *some process crashes* or votes 0

Implementing NBAC

- In the class:
  - NBAC = beb + ucons + P
- Is P really necessary?
  - NBAC4 mentions "some process"
  - hence, no (there are weaker FDs which solve NBAC)

?P

- **Anonymous Completeness:** If some process crashes, then there is a time after which every correct process permanently detects a crash.
- **Anonymous Accuracy:** No crash is detected unless some process crashes.

Outline of the idea

- Every process casts its vote
- Waits for all other votes, or a crash from ?P
- If there was a crash, or there is a vote “no”
  - run *consensus* with “abort”
  - else, run *consensus* with “commit”
- Implement consensus

Pseudo code (book)

Implements:
    NonBlockingAtomicCommit, instance nbac.
Uses:
    BestEffortBroadcast, instance beb;
    UniformConsensus, instance uc;
    EventuallyPerfectFailureDetector, instance ?P.

```
upon event  nbac, Init  do
    voted := ∅;
    proposed := FALSE;
```

Pseudo code (book)

```
upon event <?P,Crash> do
    if proposed = FALSE then
        trigger  <uc, Propose | ABORT>;
        proposed := TRUE;

upon event  <nbac, Propose | v> do
    trigger <beb, Broadcast | v>;
```

Pseudo code (book)

```
upon event <beb, Deliver | p, v > do
    if v = ABORT ∧ proposed = FALSE then
        trigger <uc, Propose | ABORT>;
        proposed := TRUE;
    else
        voted := voted ∪ {p};
        if voted = Π ∧ proposed = FALSE do
            trigger <uc, Propose | COMMIT>;
            proposed := TRUE;

upon event <uc, Decide | decided> do
    trigger <nbac, Decide | decided>;
```

we wait for all processes here!
the trick is that we won't wait forever, due to the eventual failure detector

Consensus

- FLP: Impossible to solve in asynchronous environment, when one process may crash
  - can be circumvented by using Failure Detectors and correct majority
- In the class:
  - solving Consensus using P, and <>P

Can we use ◇S?

- Short answer: "yes"
- Long answer follows...

◊S

- **Strong Completeness:** Eventually, every process that crashes is permanently suspected by every process.
- **Eventual Weak Accuracy:** Eventually, *some correct process* is *never* suspected.
  - we can have a unique correct process recognized by other processes (hint, hint)

Outline of the idea

- There is a correct process which is not suspected by anyone
  - hence, if that process is the leader during some time, it can make a decision, and use reliable broadcast to disseminate

Algorithm

- Use rotating coordinator
- Asynchronous "rounds"
  - all messages are either to, or from the coordinator
  - each round has 4 phases

Algorithm

- Phase I: Every node sends the estimate to *the coordinator*, timestamped with the round in which it adopted the value
- Phase II: *the coordinator* waits for majority of estimates
  - picks the one with the highest timestamps
  - broadcasts the new estimate

Algorithm

- Phase III:
  - if a node suspect *the coordinator,* it *nacks* the estimate
  - otherwise, it *adopts* the estimate, and *acks* it
- Phase IV:
  - the coordinator waits for majority of *responses*
  - if the coordinator gets a majority of acks, it rbcasts the decision

Pseudo code: Init

Implements:
 UniformConsensus, instance uc.
Uses:
 EventuallyStrongFailureDetector, instance $\diamond S$;
 BestEffortBroadcast, instance beb;
 UniformReliableBroadcast, instance urb.

upon event <uc, Init> do
 round := 1; suspected := $\varnothing$;
 estimate := $\perp$;
 votes := $[\perp]^N$; estimates := $[\perp]^N$;

Pseudo code: Phase I

```
upon event  <uc, Propose | v>  such that proposal = ⊥
do
    estimate := v;
    begin_round();

function begin_round() do
    trigger <beb, Broadcast|[ESTIMATE,tsp,estimate]>
```

Pseudo code: Phase II

```
upon event <beb,Deliver|p,[ESTIMATE,r,ts,v]> such that r=round
do //only leader
    est[p] =(ts,v)

upon event #(est) > N/2 do // only leader
    (ts,estimate) := highest(est);
    trigger <beb,Broadcast|[PROPOSE,round,estimate]>
```

Pseudo code: Phase III

```
upon event <beb, Deliver| p, [PROPOSE,r,v]>
                such that leader(round) = p do
    estimate := v;
    tsp := r;
    trigger <beb, Broadcast|[ACK,r]>;
    round = round+1;
    begin_round();
```

Pseudo code: Phase III

```
upon event  <◊S, Suspect | p>  do
    suspected := suspected ∪ {p};
    if leader(round) = p then
       trigger <beb, Broadcast|[NACK,round]>;
       round = round+1;
       begin_round();

upon event  <◊S, Restore | p  do
    suspected := suspected \ {p};
```

Pseudo code: Phase IV

```
upon event <beb, Deliver|p, [ACK,r]> such that r=round
do // only leader
    votes[p] := ack;

upon event <beb, Deliver|p, [NACK,r]> such that r=round
do // only leader
    votes[p] := nack;

upon #(votes) > N/2 do // only leader
    if |v: v in votes ∧ v = ack| > N/2 then
        trigger <urb, Broadcast|[DECIDED, estimate]>;

upon <urb, Deliver|[DECIDED, v]> do
    trigger <uc, Decide|v>;
```