Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.826 – Principles of Computer Systems

Handout 45 November 17, 1993

Problem Set #10

Due date: November 24, 1993

Problem 1.

The module NPrC below is a description of the presumed-commit optimization of the two-phase commit protocol from the paper by Lampson and Lomet. Your job is to give the abstraction function and rep invariant required to prove it correct. You should also sketch the proof by enumerating the actions and showing how each is explained by an appropriate sequence of actions in the specification for AtomicCommitment (which can be found in Handout ??). (Note: the invariant will be needed to be able to show that actions do what they are supposed to.) You do not have to give a proof for your invariant.

```
MODULE NPrC[P]     % P is the type of participant names

% We model the NPrC protocol for a single transaction.
% The coordinator has a stable variable prA that represents
% whether the transaction's tid is in the range tid(l) to tid(h);
% prA is true iff the tid is in the range, and is changed to false
% to represent advancing tid(l) past the transaction's tid (i.e.,
% switching to presumed commit for the transaction).

TYPE
   PS = SET P                             % set of participants.
   S  = ENUM[idle, active,                % status of transaction.
             prepared,                    % only for participant.
             committed, aborted]
   M  = [p, k]                            % message to or from participant p.
   K  = ENUM[Prepare, Commit, Abort,      % kind of message.
             voteYes, voteNo, Ack, Query]
   MS = SET M
   D  = ENUM[commit, abort]
   DN = D + Null
```

```
VAR
   ms := {}                % messages; can be lost or duplicated.
   allowed: P -> DN := {*->nil}
   outcome: P -> DN := {*->nil}

APROC Allow(p, d) = << allowed(p) = nil => allowed(p) := d [*] SKIP >>
FUNC Outcome(p) -> DN = RET outcome(p)

APROC Send(p,k) = << ms := ms ++ M{p:=p,k:=k} >>
FUNC Receive(p,k) -> Bool = RET (M{p:=p,k:=k} IN ms)

THREAD Lose() = DO << VAR m | m IN ms => ms := ms -- m [] SKIP >> OD

THREAD Coordinator() =
   VAR
      % Stable.
      prA: Bool := true,  % true if presumed abort.
      s := active,

      % Volatile (coordinator's volatile database).
      sv: S := active,    % volatile record of status.
      ps := ...,          % cohorts for transaction; initialized to set
                          %  of all cohorts (but forgotten in a crash).
      yesVotes: PS := {}  % number of yes votes received.
      acks: PS := {}      % cohorts that have acked outcome.
   | DO <<

   % Crash.  Note that if prA is true and s # committed then the
   % aborted status must be remembered forever.
   yesVotes := 0; acks := {}; ps := {};
   BEGIN prA /\ s # committed => sv := aborted
      [*] sv := idle END

   % Send prepare message.
   [] s = active => VAR p | Send(p,Prepare)

   % Receive yes vote.
   [] VAR p | Receive(p,voteYes) =>  yesVotes := yesVotes ++ p

   % Commit.
   [] yesVotes = ps /\ ps # {} => s := committed; sv := committed;
```

```
   % Abort (on timeout or no vote).
   [] sv = active => sv := aborted

   % Log abort stably.
   [] sv = aborted => s := aborted

   % Send outcome.
   [] sv = committed => VAR p | p IN ps => Send(p,Commit)
   [] sv = aborted => VAR p | p IN ps => Send(p,Abort)

   % Receive ack.
   [] VAR p | Receive(p,Ack) => acks := acks ++ p

   % Receive query; respond with outcome
   [] VAR p | Receive(p,Query) => BEGIN
           sv = idle \/ sv = committed => Send(p,Commit)
        [] sv = aborted => Send(p,Abort) END

   % Change to presumed commit.
   [] sv = committed \/
        (sv = aborted /\ acks = ps /\ ps # {}) =>
            prA := false

   % Garbage collect volatile database.
   [] prA = false => sv := idle; ps := {}; yesVotes := 0; acks := {}

   >> OD

THREAD Participant(p) =   % one for each participant.
   VAR
      % Stable.
      s := active,

   | DO <<

   % Crash.
   [] s = active => s := idle

   % Receive prepare message (prepare only if active).
   [] Receive(p,Prepare) /\ s = active /\ allowed(p) = commit =>
        s := prepared
```

```
    % Send vote message.
    [] s = prepared => Send(p,voteYes)
    [] s = idle \/ (s = active /\ allowed(p) = abort) => Send(p,voteNo)

    % Send query message.
    [] s = prepared => Send(p,Query)

    % Receive commit.
    [] Receive(p,Commit) => s := committed; outcome(p) := commit

    % Receive abort.
    [] Receive(p,Abort) => s := aborted; outcome(p) := abort

    % Send ack.
    [] s = aborted => Send(p,Ack)

    >> OD

END NPrC
```

Problem 2.

This problem is about the clock-based reliable message protocol C described in section 10.7, pages 269-273, of the handout on reliable messages.

(A)

The abstraction function from C to the generic protocol G is given at the top of page 269 (ignore the column that deals with the H protocol). It has an obvious error—what is it?

(B)

Suppose that the action `progress(i)` (in Table 10.5) that represents the passage of real time is changed to

```
 progress(i) = now < i /\ |i - times| < eps /\ |i - timer| < eps =>
      now := i
```

Explain informally what specification the changed protocol implements.

(C)

With the change described in part (B), which invariants no longer hold, and why?

(D)

It is proposed to set the parameter beta to 20 minutes in order to minimize the frequency of increase-high actions, each of which causes a stable storage write. Why is this probably not a good idea?