Csc72010

Network Protocols and Distributed Algorithms

Assignment 2

Due March 2, 2006

Objectives:

Understand impact of different reliability techniques on performance
Model important protocols (ARQ, STP).
Simple proofs of invariants

Instructions:
Please indicate the method used or give diagrams to illustrate computations. This permits partial credit if your method is close to right but your computation is wrong.
Please keep answers concise but clear. You will get more sympathy if you explain what you're doing clearly but we don't have to read volumes to grade papers.
Please simulate automata if at all possible. This is not required in question 2, but strongly recommended. I doubt that you will get the automaton right without simulation.
Copying from code in the book, in the manual, in lectures, or in handouts is permitted, as long as you credit the source. No other sources are permitted. We will question idential code unless we know it has a shared, legitimate source.

Questions:

1. How long does a take to transfer a 10-MB file over a link having an RTT of 100 ms and a frame size of 1500 bytes, in each of the following cases. Hint: approximate answers to a similar problem, problem 6, are in the back of the book. Better answers to 6a and 6b are in the attached diagrams.
   Answer:
   a. *The bandwidth is 10Mbps and frames can be sent continuously as long as none is lost. [Use this for part e: If one is lost, the previous 666 frames must be re-sent.]*
   *Transmit time = 80*1024*1024/10000000=8.388608 seconds*
   Propagation time = 50 ms
   Total transfer time = 8.438608 seconds

   b. *The bandwidth is 10Mbps and after sending each frame, we must wait an RTT to send the next (e.g., stop and wait).*
   *Transmit time per frame = 8*1500/10000000=1.2 ms*
   RTT = 100 ms (this covers the propagation time)
   Total transfer time per frame = 101.2 ms
   *Total number of frames = 10*1024*1024/1500 = 6990 + a runt frame of 760 bytes*
   *[6990*1500=10485000, so 760 bytes are left over]*

*For all but the last frame, total transfer time = 6990*101.2 ms = 707.388 seconds*
The last frame takes total time = transmit time+propagation time = 6080/10000000 + 50 ms = .608 ms + 50 ms = 50.608 ms = .050608 secs
So the total for all frames is 707.438608 seconds.

c. *The bandwidth is infinite, ie transmit time is 0, and we can send 30 frames, then wait an RTT for an ACK. [Use this for part e: If the ACK doesn't arrive, we send the previous 30.]*
This is just like sending a 45000 byte frame, then waiting an RTT.
*Total number of frames is: 10*1024*1024/45000=233 frames, plus a runt frame of size 760 bytes.*
Time for each of the first 233 frames is .1 seconds
Total time for the first 233 frames is 233.1 = 23.3 seconds*
The final frame takes .050 seconds
Total time = 23.35 seconds

d. *The bandwidth is infinite. During the first RTT, we can send 1 frame (2^{1-1}). During the second RTT, we can send the 2 frames (2^{2-1}). During the third RTT, we can send 4 frames (2^{3-1}) . [For part e: If a frame is lost, re-send the previous batch and cut the number of frames by half in the next RTT.]*
How many rounds of frames do we send?
We send for n rounds, where $2^n-1 = 1+2^1+...+2^{n-1} <= 6991 <= 1+2^1+...+2^n = 2^{n+1}-1$. Thus n=12 ($2^{12}-1=4095$ and $2^{13}-1=8191$). The last round contains 6991-4095=2896 frames.
Each round except the last takes 100 ms; the last takes 50 ms. So the total time is 1250 ms or 1.25 seconds.

e. *(Extra credit) If 1 frame in each 100,000 is lost, how does that affect the worst-case time for each of the above questions? Alternatively, determine the expected time to transfer the file.*
    i. *The simple-minded answer is: If a frame is lost, must send an extra 666*1500 bytes = 999000 bytes = 7992000 bits. This adds .7992 seconds o the transmit time. Total transfer time = 8.438608 + .7992 seconds =9.237808 seconds.*
    However... if we assume that we're running go-back-n and filling the pipe to capacity as we do it, so that when a frame is dropped we suffer a penalty of n frames, then for a 10Mbps bandwidth and .05 second delay, the pipe holds 500,000 bits or 41.666... frames in the forward direction and we need to allow the same number of acks in the reverse direction. So, the window needs to hold 84 (83.3333) frames, and when we discover that a frame has been lost we must go back 84. The 666 seems to have been the result of a flakey computation on my part. So, the actual penalty should not be so bad.

Note that this requires assuming that frames can only be dropped, not reordered or duplicated. The ack for frame 1 should arrive before we are ready to send frame 85, so we go back 84 if it doesn't arrive. And so forth.

ii. *Send the extra frame, requiring an extra 101.2 ms. Total transfer time 708.388658 +.1012 seconds = 708.489858 seconds*

iii. *Time for the extra 30 frames is .1 seconds, so total transfer time is 23.45 seconds.*

iv. *Two or three extra batches, depending on how long it takes before the error happens:*
First frame: $1+1+1+2+...+2^{n-1}+2896$, where n=12 – so, two extra batches (the two batches of one frame each at the beginning)
Frame 2 or 3: $1+2+2+1+2+...+2^{n-1}+2893$, where n=12 – so, three extra batches (of 2, 2, and 1 frames, respectively)
Frames 4, 5, 6, or 7: $1+2+4+4+2+4+...+2^{11}+2890$
The number of extra batches is 3, until we get to some i where all 2896 of the last batch of frames are taken care of in the sum $2^{i-1} + 2^i$, ie, $3*2^{i-1} > 2896$. This happens for i=10.
A frame in the next to last batch: $1+2+...+2^{11}+2^{11} + 2^{10} + 1872$ – since we have to send only 1024 in the batch after the 2^{11} batch, we have some extras left over, so there are 2 extra batches.
A frame in the final batch: $1+2+4+...+2^{11}+(2896)+(2896)$ – just one extra batch!

2. Define one or two IOAs for the Sliding Window Protocol with Cumulative Acknowledgment.

   Answer: *See ioa at end.*

   The data part of the message may be defined as an Int, for simplicity.
   Define a tuple type for messages, including sequence number, acknowledgment number (next expected sequence number), and data.
   You may either initialize the state with a buffer of messages to send or create an init action that initializes the state. To test, make sure you test with the initial array much larger than the window size.
   Note that in simulating, you need only two processes connected by a lossy, reordering, duplicating channel (you can compose a reordering channel with a lossy channel and a duplicating channel to achieve this).

   Ideally, you will define a single two-way IOA, but you will get 75% credit for defining a sending IOA and a receiving IOA. I recommend trying the two-way IOA, but if you find that hard another possible strategy is to define the two IOA's first, then combine them if you have time.

3. State and prove the invariant relating window size (SWS), last acknowledgment

received (LAR), and next frame (NEXT) to be sent: NEXT <= LAR+SWS+1. You can test the invariant using the simulation.

Answer:

It seems to be stated above :-). The reasoning is by induction, using the code. In my code, NEXT is represented by the variable seq, LAR by lar, and SWS by N, so the invariant is seq<=lar+N+1.

***Base case**: Initially, seq = 1, and for window size N>=1, seq <= lar+N.*

***Induction hypothesis**: In every execution of fewer than k steps, seq <= lar+N+1.*

Induction step:

Case 1: The action at step k is debug(s,t): No variables change and therefore the invariant is still true.

Case 2: The action at step k is send(m,p,j): The action adds 1 to seq if seq <=lar+N. Thus after this step, seq can be at most lar+N+1.

Case 3: The action at step k is receive(m,i,p):

The first statement (if (m.ack-1) > lar then lar := m.ack-1 fi;) can change lar, but only by increasing it. If the invariant is true, it will continue to be true. No other statements change either seq or lar.

4. The attached IOA is an approximate model of the Cisco Spanning Tree Protocol.

   An important part of the algorithm is to send a BPDU every 2 seconds for a number of rounds. Since we can't keep time in this IOA model, the IOA models this by having the scheduler fire the actions in rounds. We only are only interested in the executions that this scheduler can produce.

   Note also that the number of rounds that a process spends in `electingRoot` state (i.e., leader election) is a parameter of the process, and that the process changes state to `designatingPorts` after this many rounds have passed. It stays in `designatingPorts` for only one round.

   a. Assume that you know the configuration of the network, i.e., the graph representing the connectivity of the bridges. How many rounds should the processes stay in `electingRoot`? Give a general rule for the number of rounds.
      Answer: *Number of rounds = diameter of the network.*

   b. [Omit this for now] Prove that the set of bridges in the network together with the edges incident on a `rootPort` is a spanning tree, after running the STP algorithm on the network, assuming that the number of rounds satisfies the rule in part a.

IOA for sliding window protocol (question 2)

```
axioms NonDet

type Message = tuple of seq:Int, ack:Int, msg:Int

automaton SWP(p:Int, N:Int, msg:Array[Int,Int])

signature
        output send(m:Message, const p, j:Int), debug(s:Int, t:Int)
        input receive(m:Message, i:Int, const p)

states
      tosend:Array[Int,Int] := msg,              % data to be sent
        received:Array[Int,Int] := constant(-1),   % data received
        lar:Int := 0,                            % last ack received
        seq:Int := 1,                             % next frame to send
        lfr:Int := 0,                            % last frame received
                                        % with no prior holes
                                        % in receive buffer
      newAck:Bool := false,                      % true if need to send ack
        ack:Int := 1                             % next expected frame

transitions

      output debug(s, t)                         % for debugging purposes

        output send(m, p, j)
      pre
          % we're done when lar >= 6 and lfr >= 6 and seq > 6 and ack > 6
          (lar < 6 \/ lfr < 6 \/ (seq-1) < 6 \/ (ack-1) < 6) /\
          % send the next Int in "tosend" if seq <= lar+N
          ((seq<=(lar+N) /\ m=[seq,ack,tosend[seq]]) \/
          % resend the first Int after lar if seq > lar+N
           (seq>(lar+N) /\ m=[lar+1,ack,tosend[lar+1]]))
        eff
        % if we sent a new message, increment seq
        if seq <= (lar+N) then
          seq := seq+1
        fi;

        input receive(m, i, p)
        eff
        % functioning as sender
        % if we received a larger ack than any previous one, increment lar
           if (m.ack-1) > lar then lar := m.ack-1 fi;

        % functioning as receiver
        % if message is inside legal range, put it in receive buffer
           if (m.seq > lfr /\ m.seq <= (lfr+N) /\ m.msg ~= -1) then
          received[m.seq] := m.msg
```

```
        fi;

        newAck := false;        % assume no change in message to be acked
        % check to see if a new message should be acked
        %    the following should be a loop from lfr+1 to lfr+N, looking
        %    for the largest seq with no previous "holes" in the receive
        %    buffer;
        %    the simulator doesn't seem to execute the loop correctly.
        if received[lfr+1] ~= -1 then
          newAck := true;
          if received[lfr+2] ~= -1 then
            if received[lfr+3] ~= -1 then
              lfr := lfr+3;
            else
              lfr := lfr+2;
            fi
          else
            lfr := lfr+1;
          fi
        fi;           % keep adding "if's" up to lfr+N as necessary
        % set the value of the ack
        ack := lfr+1

% this is a reordering channel
automaton MsetChannel(i, j: Int)
  signature
    input send(m: Message, const i, const j)
    output receive(m: Message, const i, const j)

  states buffer: Mset[Message] := {}

  transitions
    input send(m, i, j)
      eff buffer := insert(m, buffer)

    output receive(m, i, j)
      pre m \in buffer
      eff buffer := delete(m, buffer)


%this is a lossy channel (see input send)
automaton LossyChannel(i, j: Int)

signature
  input send(m: Message, const i, const j)
  output receive(m: Message, const i, const j)

states
  buffer: Seq[Message] := {}

transitions
```

```
  input send(m, i, j)
    eff
      % throw a 3-sided die to decide whether to lose the message
      if randomInt(1,5)<5 then buffer := buffer |- m fi

  output receive(m, i, j)
    pre buffer ~= {} /\ m = head(buffer)
    eff buffer := tail(buffer)

%this is a lossy channel (see input send)
automaton DuplicatingChannel(i, j: Int)

signature
  input send(m: Message, const i, const j)
  output receive(m: Message, const i, const j)

states
  buffer: Seq[Message] := {}

transitions
  input send(m, i, j)
    eff
      % throw a 3-sided die to decide whether to lose the message
      buffer := buffer |-m;
      if randomInt(1,3)=3 then buffer := buffer |- m fi

  output receive(m, i, j)
    pre buffer ~= {} /\ m = head(buffer)
    eff buffer := tail(buffer)

%this is a reliable FIFO channel
automaton ReliableChannel(i, j:Int)
 signature
   input send(m: Message, const i, const j)
   output receive(m: Message, const i, const j) , debug(s:Int)
 states
   buffer: Seq[Message] := {}
 transitions
   input send(m, i, j)
     eff buffer := buffer |- m
   output receive(m, i, j)
     pre m = head(buffer)
     eff buffer := tail(buffer)
   output debug(s)

automaton Network(s:Array[Int,Int])
 components C[m:Int,n:Int]: LossyChannel(m,n)
          where 1 <= m /\ m <= 2 /\ 1 <= n /\ n <= 2;
          P[i:Int]:SWP(i, 3,
            assign(assign(assign(assign(assign(assign(constant(-
1),1,1),2,2),3,3),4,4),5,5),6,6) )
```

```
            where 1 <= i /\ i <= 2

schedule
states
  rndm:Int,
  prcs:Int,
  chan:Int,
  message:Message,
  x:Message
with
   C12 = C[1,2],
   C21 = C[2,1],
   P1 = P[1],
   P2 = P[2]
do

  while true do
    % let i represent Pi for i<=2; i is Ci,i+1 for i>3
    rndm := randomInt(1,4);
    %fire output P1.debug(rndm);
    if rndm <= 2 then
      prcs := rndm;
      if prcs=1 then
        if P1.lar < 6 \/ P1.lfr < 6 then
           %fire output P1.debug(P1.seq, P1.lar);
           if P1.seq <= (P1.lar+3) then
             message := [P1.seq, P1.ack, P1.tosend[P1.seq]];
             fire output P1.send(message, 1, 2)
           else
             message := [P1.lar+1, P1.ack, P1.tosend[P1.lar+1]];
             fire output P1.send(message, 1, 2)
           fi
        fi
      elseif prcs=2 then
        if P2.lar < 6 \/ P2.lfr < 6  then
           %fire output P2.debug(P2.seq, P2.lar);
           if P2.seq <= (P2.lar+3) then
             message := [P2.seq, P2.ack, P2.tosend[P2.seq]];
             fire output P2.send(message, 2, 1)
           else
             message := [P2.lar+1, P2.ack, P2.tosend[P2.lar+1]];
             fire output P2.send(message, 2, 1)
           fi
        fi
      fi;
      if C12.buffer ~= {} then
        message := head(C12.buffer);
        %fire output C12.debug(message.msg);
        fire output C12.receive(message,1,2)
      fi;
      if C21.buffer ~= {} then
```

```
        message := head(C21.buffer);
        %fire output C21.debug(message.msg);
        fire output C21.receive(message,2,1)
      fi
    fi
  od
od
```