Project 3: Implementing an Algorithm

Due Apr. 29th, 2014

Hand in: Electronic submission of entire project (source, compiled, excutable files, etc.) on Canvas. Please zip up your whole project directory and submit the zip file.

Overview

In this lab, you will be writing a "distributed" set of procedures that implement a distributed asynchronous distance vector routing for the network shown in Figure Lab.4-1.

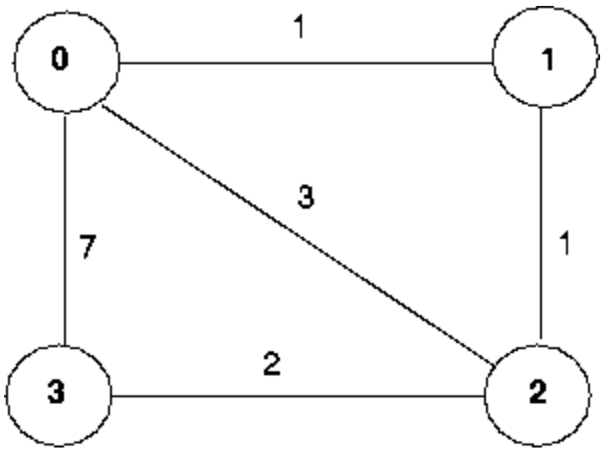

Figure Lab.4-1: Network topology and link costs for DV routing lab

The Basic Assignment

The routines you will write For the basic part of the assignment, you are to write the following routines which will ``execute'' asynchronously within the emulated environment that we have written for this assignment.

For node 0, you will write the routines:

- `rtinit0()` This routine will be called once at the beginning of the emulation. `rtinit0()` has no arguments. It should initialize the distance table in node 0 to reflect the direct costs of 1, 3, and 7 to nodes 1, 2, and 3, respectively. In Figure 1, all links are bi-directional and the costs in both directions are identical. After initializing the distance table, and any other data structures needed by your node 0 routines, it should then send its directly-connected neighbors (in this case, 1, 2 and 3) the cost of it minimum cost paths to all other network nodes. This minimum cost information is sent to neighboring nodes in a *routing packet* by calling the routine `tolayer2()`, as described below. The format of the routing packet is also described below.
- `rtupdate0(struct rtpkt *rcvdpkt)`. This routine will be called when node 0 receives a routing packet that was sent to it by one if its directly connected neighbors. The parameter `*rcvdpkt` is a pointer to the packet that was received.

`rtupdate0()` is the "heart" of the distance vector algorithm. The values it receives in a routing packet from some other node *i* contain *i*'s current shortest path costs to all other network nodes. `rtupdate0()` uses these received values to update its own distance table (as specified by the distance vector algorithm). If its own minimum cost to another node changes as a result of the update, node 0 informs its directly connected neighbors of this change in minimum cost by sending them a routing packet. Recall that in the distance vector algorithm, only directly connected nodes will exchange routing packets. Thus nodes 1 and 2 will communicate with each other, but nodes 1 and 3 will node communicate with each other.

As we saw in class, the distance table inside each node is the principal data structure used by the distance vector algorithm. You will find it convenient to declare the distance table as a 4-by-4 array of `int`'s, where entry `[i,j]` in the distance table in node 0 is node 0's currently computed cost to node i via direct neighbor j. If 0 is not directly connected to *j,* you can ignore this entry. We will use the convention that the integer value 999 is ``infinity.''

Figure Lab.4-2 provides a conceptual view of the relationship of the procedures inside node 0.

Similar routines are defined for nodes 1, 2 and 3. Thus, you will write 8 procedures in all:
`rtinit0(), rtinit1(), rtinit2(), rtinit3(),rtupdate0(), rtupdate1(), rtupdate2(), rtupdate3()`

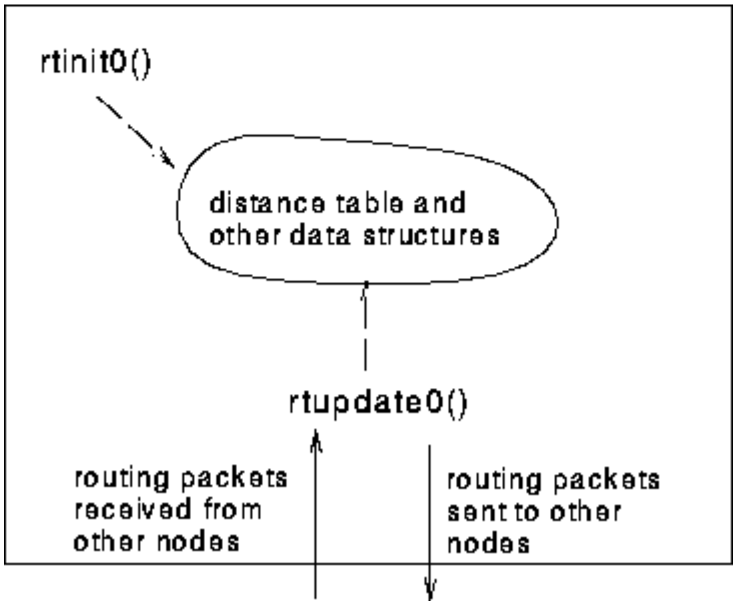
rtinit0()
distance table and other data structures
rtupdate0()
routing packets received from other nodes
routing packets sent to other nodes

Figure Lab.4-2: Relationship between procedures inside node 0

Software Interfaces

The procedures described above are the ones that you will write. We have written the following routines that can be called by your routines:

`tolayer2(struct rtpkt pkt2send)`

where `rtpkt` is the following structure, which is already declared for you. The procedure `tolayer2()` is defined in the file prog3.c

```
extern struct rtpkt {
  int sourceid;  /* id of node sending this pkt, 0, 1, 2, or 3 */
  int destid;    /* id of router to which pkt being sent
```

```
                        (must be an immediate neighbor) */
  int mincost[4];        /* min cost to node 0 ... 3 */
  };
```

Note that `tolayer2()` is passed a structure, not a pointer to a structure.

`printdt0()`

will pretty print the distance table for node 0. It is passed a pointer to a structure of type `distance_table`. `printdt0()` and the structure declaration for the node 0 distance table are declared in the file `node0.c`. Similar pretty-print routines are defined for you in the files `node1.c, node2.c node3.c.`

The simulated network environment

Your procedures `rtinit0(), rtinit1(), rtinit2(), rtinit3()` and `rtupdate0(), rtupdate1(), rtupdate2(), rtupdate3()` send routing packets (whose format is described above) into the medium. The medium will deliver packets in-order, and without loss to the specified destination. Only directly-connected nodes can communicate. The delay between is sender and receiver is variable (and unknown).

When you compile your procedures and my procedures together and run the resulting program, you will be asked to specify only one value regarding the simulated network environment:

- **Tracing.** Setting a tracing value of 1 or 2 will print out useful information about what is going on inside the emulation (e.g., what's happening to packets and timers). A tracing value of 0 will turn this off. A tracing value greater than 2 will display all sorts of odd messages that are for my own emulator-debugging purposes.

A tracing value of 2 may be helpful to you in debugging your code. You should keep in mind that *real* implementors do not have underlying networks that provide such nice information about what is going to happen to their packets!

The Basic Assignment

You are to write the procedures `rtinit0(), rtinit1(), rtinit2(), rtinit3()` and `rtupdate0(), rtupdate1(), rtupdate2(), rtupdate3()` which together will implement a distributed, asynchronous computation of the distance tables for the topology and costs shown in Figure 1.

You should put your procedures for nodes 0 through 3 in files called node0.c, ... node3.c. You are **NOT** allowed to declare any global variables that are visible outside of a given C file (e.g., any global variables you define in `node0.c.` may only be accessed inside `node0.c`). This is to force you to abide by the coding conventions that you would have to adopt is you were really running the procedures in four distinct nodes. To compile your routines: **`cc -w prog3.c node0.c node1.c node2.c node3`**. Prototype versions of these files prog3.c node0.c, node1.c, node2.c, node3.c are listed in the end of the handout.

This assignment can be completed on any machine supporting C. It makes no use of UNIX features.

For your sample output, your procedures should print out a message whenever your `rtinit0(), rtinit1(), rtinit2(), rtinit3()` or `rtupdate0(), rtupdate1(), rtupdate2(), rtupdate3()` procedures are called, giving the time (available via my global variable `clocktime`). For `rtupdate0(), rtupdate1(), rtupdate2(), rtupdate3()` you should print the identity of the sender of the routing packet that is being passed to your routine, whether or not the distance table is updated, the contents of the distance table (you can use my pretty-print routines), and a description of any messages sent to neighboring nodes as a result of any distance table updates.

The sample output should be an output listing with a TRACE value of 2. Highlight the final distance table produced in each node. Your program will run until there are no more routing packets in-transit in the network, at which point our emulator will terminate.

Code

Prog3.c code

```
#include <stdio.h>

#define LINKCHANGES 1
/* ******************************************************************
Programming assignment  : implementing distributed, asynchronous,
                          distance vector routing.

THIS IS THE MAIN ROUTINE.  IT SHOULD NOT BE TOUCHED AT ALL BY STUDENTS!

*********************************************************************/


/* a rtpkt is the packet sent from one routing update process to
   another via the call tolayer3() */
struct rtpkt {
  int sourceid;       /* id of sending router sending this pkt */
  int destid;         /* id of router to which pkt being sent
                         (must be an immediate neighbor) */
  int mincost[4];    /* min cost to node 0 ... 3 */
  };

int TRACE = 1;              /* for my debugging */
int YES = 1;
int NO = 0;

creatertpkt( initrtpkt, srcid, destid, mincosts)
struct rtpkt *initrtpkt;
int srcid;
int destid;
int mincosts[];

{
  int i;
  initrtpkt->sourceid = srcid;
  initrtpkt->destid = destid;
  for (i=0; i<4; i++)
    initrtpkt->mincost[i] = mincosts[i];
}


/*****************************************************************
```

```
***************** NETWORK EMULATION CODE STARTS BELOW ***********
The code below emulates the layer 2 and below network environment:
  - emulates the tranmission and delivery (with no loss and no
    corruption) between two physically connected nodes
  - calls the initializations routines rtinit0, etc., once before
    beginning emulation

THERE IS NOT REASON THAT ANY STUDENT SHOULD HAVE TO READ OR UNDERSTAND
THE CODE BELOW.  YOU SHOLD NOT TOUCH, OR REFERENCE (in your code) ANY
OF THE DATA STRUCTURES BELOW.  If you're interested in how I designed
the emulator, you're welcome to look at the code - but again, you
should have
to, and you defeinitely should not have to modify
******************************************************************/

struct event {
   float evtime;           /* event time */
   int evtype;             /* event type code */
   int eventity;           /* entity where event occurs */
   struct rtpkt *rtpktptr; /* ptr to packet (if any) assoc w/ this
event */
   struct event *prev;
   struct event *next;
 };
struct event *evlist = NULL;   /* the event list */

/* possible events: */
#define  FROM_LAYER2     2
#define  LINK_CHANGE     10

float clocktime = 0.000;


main()
{
   struct event *eventptr;

   init();

   while (1) {

        eventptr = evlist;            /* get next event to simulate */
        if (eventptr==NULL)
           goto terminate;
        evlist = evlist->next;        /* remove this event from event
list */
        if (evlist!=NULL)
           evlist->prev=NULL;
        if (TRACE>1) {
          printf("MAIN: rcv event, t=%.3f, at %d",
                          eventptr->evtime,eventptr->eventity);
          if (eventptr->evtype == FROM_LAYER2 ) {
            printf(" src:%2d,",eventptr->rtpktptr->sourceid);
            printf(" dest:%2d,",eventptr->rtpktptr->destid);
            printf(" contents: %3d %3d %3d %3d\n",
              eventptr->rtpktptr->mincost[0], eventptr->rtpktptr-
>mincost[1],
              eventptr->rtpktptr->mincost[2], eventptr->rtpktptr-
>mincost[3]);
            }
          }
        clocktime = eventptr->evtime;    /* update time to next event
time */
        if (eventptr->evtype == FROM_LAYER2 ) {
            if (eventptr->eventity == 0)
              rtupdate0(eventptr->rtpktptr);
             else if (eventptr->eventity == 1)
              rtupdate1(eventptr->rtpktptr);
             else if (eventptr->eventity == 2)
              rtupdate2(eventptr->rtpktptr);
             else if (eventptr->eventity == 3)
              rtupdate3(eventptr->rtpktptr);
             else { printf("Panic: unknown event entity\n"); exit(0); }
          }
        else if (eventptr->evtype == LINK_CHANGE ) {
            if (clocktime<10001.0) {
              linkhandler0(1,20);
              linkhandler1(0,20);
              }
           else   {
              linkhandler0(1,1);
              linkhandler1(0,1);
              }
          }
          else
             { printf("Panic: unknown event type\n"); exit(0); }
        if (eventptr->evtype == FROM_LAYER2 )
          free(eventptr->rtpktptr);        /* free memory for packet,
if any */
        free(eventptr);                    /* free memory for event
struct   */
      }


terminate:
   printf("\nSimulator terminated at t=%f, no packets in medium\n",
clocktime);
}



init()                         /* initialize the simulator */
{
  int i;
  float sum, avg;
  float jimsrand();
  struct event *evptr;

   printf("Enter TRACE:");
   scanf("%d",&TRACE);

   srand(9999);              /* init random number generator */
   sum = 0.0;                /* test random number generator for
students */
```

```
   for (i=0; i<1000; i++)
      sum=sum+jimsrand();    /* jimsrand() should be uniform in [0,1]
*/
   avg = sum/1000.0;
   if (avg < 0.25 || avg > 0.75) {
    printf("It is likely that random number generation on your
machine\n" );
    printf("is different from what this emulator expects.  Please
take\n");
    printf("a look at the routine jimsrand() in the emulator code.
Sorry. \n");
    exit();
    }

   clocktime=0.0;                /* initialize time to 0.0 */
   rtinit0();
   rtinit1();
   rtinit2();
   rtinit3();

   /* initialize future link changes */
  if (LINKCHANGES==1)   {
   evptr = (struct event *)malloc(sizeof(struct event));
   evptr->evtime =  10000.0;
   evptr->evtype =  LINK_CHANGE;
   evptr->eventity =  -1;
   evptr->rtpktptr =  NULL;
   insertevent(evptr);
   evptr = (struct event *)malloc(sizeof(struct event));
   evptr->evtype =  LINK_CHANGE;
   evptr->evtime =  20000.0;
   evptr->rtpktptr =  NULL;
   insertevent(evptr);
   }

}

/**********************************************************************
******/
/* jimsrand(): return a float in range [0,1].  The routine below is
used to */
/* isolate all random number generation in one location.  We assume
that the*/
/* system-supplied rand() function return an int in therange [0,mmm]
*/
/**********************************************************************
******/
float jimsrand()
{
  double mmm = 2147483647;   /* largest int  - MACHINE DEPENDENT!!!!!!!!
*/
  float x;                   /* individual students may need to change
mmm */
  x = rand()/mmm;            /* x should be uniform in [0,1] */
  return(x);
}
```

```
/******************** EVENT HANDLINE ROUTINES *******/
/*  The next set of routines handle the event list   */
/*****************************************************/


insertevent(p)
   struct event *p;
{
   struct event *q,*qold;

   if (TRACE>3) {
      printf("            INSERTEVENT: time is %lf\n",clocktime);
      printf("            INSERTEVENT: future time will be %lf\n",p-
>evtime);
      }
   q = evlist;     /* q points to header of list in which p struct
inserted */
   if (q==NULL) {   /* list is empty */
        evlist=p;
        p->next=NULL;
        p->prev=NULL;
        }
     else {
        for (qold = q; q !=NULL && p->evtime > q->evtime; q=q->next)
              qold=q;
        if (q==NULL) {   /* end of list */
             qold->next = p;
             p->prev = qold;
             p->next = NULL;
             }
           else if (q==evlist) { /* front of list */
             p->next=evlist;
             p->prev=NULL;
             p->next->prev=p;
             evlist = p;
             }
           else {     /* middle of list */
             p->next=q;
             p->prev=q->prev;
             q->prev->next=p;
             q->prev=p;
             }
         }
}

printevlist()
{
  struct event *q;
  printf("--------------\nEvent List Follows:\n");
  for(q = evlist; q!=NULL; q=q->next) {
    printf("Event time: %f, type: %d entity: %d\n",q->evtime,q-
>evtype,q->eventity);
    }
  printf("--------------\n");
}
```

```
/************************ TOLAYER2 ***************/
void tolayer2(packet)
  struct rtpkt packet;

{
 struct rtpkt *mypktptr;
 struct event *evptr, *q;
 float jimsrand(),lastime;
 int i;

 int connectcosts[4][4];

 /* initialize by hand since not all compilers allow array
initilization */
 connectcosts[0][0]=0;  connectcosts[0][1]=1;  connectcosts[0][2]=3;
 connectcosts[0][3]=7;
 connectcosts[1][0]=1;  connectcosts[1][1]=0;  connectcosts[1][2]=1;
 connectcosts[1][3]=999;
 connectcosts[2][0]=3;  connectcosts[2][1]=1;  connectcosts[2][2]=0;
 connectcosts[2][3]=2;
 connectcosts[3][0]=7;  connectcosts[3][1]=999;  connectcosts[3][2]=2;
 connectcosts[3][3]=0;

 /* be nice: check if source and destination id's are reasonable */
 if (packet.sourceid<0 || packet.sourceid >3) {
   printf("WARNING: illegal source id in your packet, ignoring
packet!\n");
   return;
   }
 if (packet.destid<0 || packet.destid >3) {
   printf("WARNING: illegal dest id in your packet, ignoring
packet!\n");
   return;
   }
 if (packet.sourceid == packet.destid)  {
   printf("WARNING: source and destination id's the same, ignoring
packet!\n");
   return;
   }
 if (connectcosts[packet.sourceid][packet.destid] == 999)  {
   printf("WARNING: source and destination not connected, ignoring
packet!\n");
   return;
   }

/* make a copy of the packet student just gave me since he/she may
decide */
/* to do something with the packet after we return back to him/her */
 mypktptr = (struct rtpkt *) malloc(sizeof(struct rtpkt));
 mypktptr->sourceid = packet.sourceid;
 mypktptr->destid = packet.destid;
 for (i=0; i<4; i++)
    mypktptr->mincost[i] = packet.mincost[i];
 if (TRACE>2)  {
   printf("    TOLAYER2: source: %d, dest: %d\n             costs:",
          mypktptr->sourceid, mypktptr->destid);
   for (i=0; i<4; i++)
```

```
        printf("%d  ",mypktptr->mincost[i]);
    printf("\n");
   }

/* create future event for arrival of packet at the other side */
  evptr = (struct event *)malloc(sizeof(struct event));
  evptr->evtype =  FROM_LAYER2;   /* packet will pop out from layer3 */
  evptr->eventity = packet.destid; /* event occurs at other entity */
  evptr->rtpktptr = mypktptr;       /* save ptr to my copy of packet */

/* finally, compute the arrival time of packet at the other end.
   medium can not reorder, so make sure packet arrives between 1 and 10
   time units after the latest arrival time of packets
   currently in the medium on their way to the destination */
 lastime = clocktime;
 for (q=evlist; q!=NULL ; q = q->next)
    if ( (q->evtype==FROM_LAYER2  && q->eventity==evptr->eventity) )
      lastime = q->evtime;
 evptr->evtime =  lastime + 2.*jimsrand();


 if (TRACE>2)
     printf("    TOLAYER2: scheduling arrival on other side\n");
 insertevent(evptr);
}
```

node0. c code

```
#include <stdio.h>

extern struct rtpkt {
  int sourceid;       /* id of sending router sending this pkt */
  int destid;         /* id of router to which pkt being sent
                         (must be an immediate neighbor) */
  int mincost[4];    /* min cost to node 0 ... 3 */
  };

extern int TRACE;
extern int YES;
extern int NO;

struct distance_table
{
  int costs[4][4];
} dt0;


/* students to write the following two routines, and maybe some others
*/

void rtinit0()
{

}
```

```
void rtupdate0(rcvdpkt)
  struct rtpkt *rcvdpkt;
{

}


printdt0(dtptr)
  struct distance_table *dtptr;

{
  printf("                via     \n");
  printf("   D0 |    1     2    3 \n");
  printf("  ----|-----------------\n");
  printf("     1|  %3d   %3d   %3d\n",dtptr->costs[1][1],
         dtptr->costs[1][2],dtptr->costs[1][3]);
  printf("dest 2|  %3d   %3d   %3d\n",dtptr->costs[2][1],
         dtptr->costs[2][2],dtptr->costs[2][3]);
  printf("     3|  %3d   %3d   %3d\n",dtptr->costs[3][1],
         dtptr->costs[3][2],dtptr->costs[3][3]);
}

linkhandler0(linkid, newcost)
  int linkid, newcost;

/* called when cost from 0 to linkid changes from current value to
newcost*/
/* You can leave this routine empty if you're an undergrad. If you want
*/
/* to use this routine, you'll need to change the value of the
LINKCHANGE */
/* constant definition in prog3.c from 0 to 1 */

{
}
```

node1.c code

```
#include <stdio.h>

extern struct rtpkt {
  int sourceid;       /* id of sending router sending this pkt */
  int destid;         /* id of router to which pkt being sent
                         (must be an immediate neighbor) */
  int mincost[4];    /* min cost to node 0 ... 3 */
  };


extern int TRACE;
extern int YES;
extern int NO;

int connectcosts1[4] = { 1,  0,  1, 999 };
```

```
struct distance_table
{
  int costs[4][4];
} dt1;


/* students to write the following two routines, and maybe some others
*/


rtinit1()
{

}


rtupdate1(rcvdpkt)
  struct rtpkt *rcvdpkt;

{

}


printdt1(dtptr)
  struct distance_table *dtptr;

{
  printf("             via   \n");
  printf("   D1 |    0     2 \n");
  printf("  ----|-----------\n");
  printf("     0|  %3d   %3d\n",dtptr->costs[0][0], dtptr->costs[0][2]);
  printf("dest 2|  %3d   %3d\n",dtptr->costs[2][0], dtptr->costs[2][2]);
  printf("     3|  %3d   %3d\n",dtptr->costs[3][0], dtptr->costs[3][2]);

}



linkhandler1(linkid, newcost)
int linkid, newcost;
/* called when cost from 1 to linkid changes from current value to
newcost*/
/* You can leave this routine empty if you're an undergrad. If you want
*/
/* to use this routine, you'll need to change the value of the
LINKCHANGE */
/* constant definition in prog3.c from 0 to 1 */

{
}
```

node2.c code

```
#include <stdio.h>
```

```
extern struct rtpkt {
  int sourceid;       /* id of sending router sending this pkt */
  int destid;         /* id of router to which pkt being sent
                         (must be an immediate neighbor) */
  int mincost[4];    /* min cost to node 0 ... 3 */
  };

extern int TRACE;
extern int YES;
extern int NO;

struct distance_table
{
  int costs[4][4];
} dt2;


/* students to write the following two routines, and maybe some others
*/

void rtinit2()
{
}


void rtupdate2(rcvdpkt)
  struct rtpkt *rcvdpkt;

{

}


printdt2(dtptr)
  struct distance_table *dtptr;

{
  printf("                via     \n");
  printf("   D2 |    0     1    3 \n");
  printf("  ----|-----------------\n");
  printf("     0|  %3d   %3d   %3d\n",dtptr->costs[0][0],
         dtptr->costs[0][1],dtptr->costs[0][3]);
  printf("dest 1|  %3d   %3d   %3d\n",dtptr->costs[1][0],
         dtptr->costs[1][1],dtptr->costs[1][3]);
  printf("     3|  %3d   %3d   %3d\n",dtptr->costs[3][0],
         dtptr->costs[3][1],dtptr->costs[3][3]);
}
```

node3.c code

```
#include <stdio.h>

extern struct rtpkt {
  int sourceid;       /* id of sending router sending this pkt */
  int destid;         /* id of router to which pkt being sent
                         (must be an immediate neighbor) */
  int mincost[4];    /* min cost to node 0 ... 3 */
  };

extern int TRACE;
extern int YES;
extern int NO;

struct distance_table
{
  int costs[4][4];
} dt3;

/* students to write the following two routines, and maybe some others
*/

void rtinit3()
{
}


void rtupdate3(rcvdpkt)
  struct rtpkt *rcvdpkt;

{

}


printdt3(dtptr)
  struct distance_table *dtptr;

{
  printf("             via     \n");
  printf("   D3 |    0     2 \n");
  printf("  ----|-----------\n");
  printf("     0|  %3d   %3d\n",dtptr->costs[0][0], dtptr->costs[0][2]);
  printf("dest 1|  %3d   %3d\n",dtptr->costs[1][0], dtptr->costs[1][2]);
  printf("     2|  %3d   %3d\n",dtptr->costs[2][0], dtptr->costs[2][2]);

}
```