Internet Protocols: Lab Assignment I: IP Fragmentation and Reassembly

Due (ON-CAMPUS/NON-DELAYED): February 14th, THURSDAY, 2002 at the beginning of class.
Due (DELAYED): February 20th, WEDNESDAY, 2002

1 Goals

- To understand the design of the fragmentation and reassembly algorithms in the IP protocol.
- To implement the fragmentation and reassembly modules for a simple IP-like network layer.
- To understand the effect of fragmentation and measure the performance under various network conditions.
- To compare a UDP/IP service with a TCP/IP service.

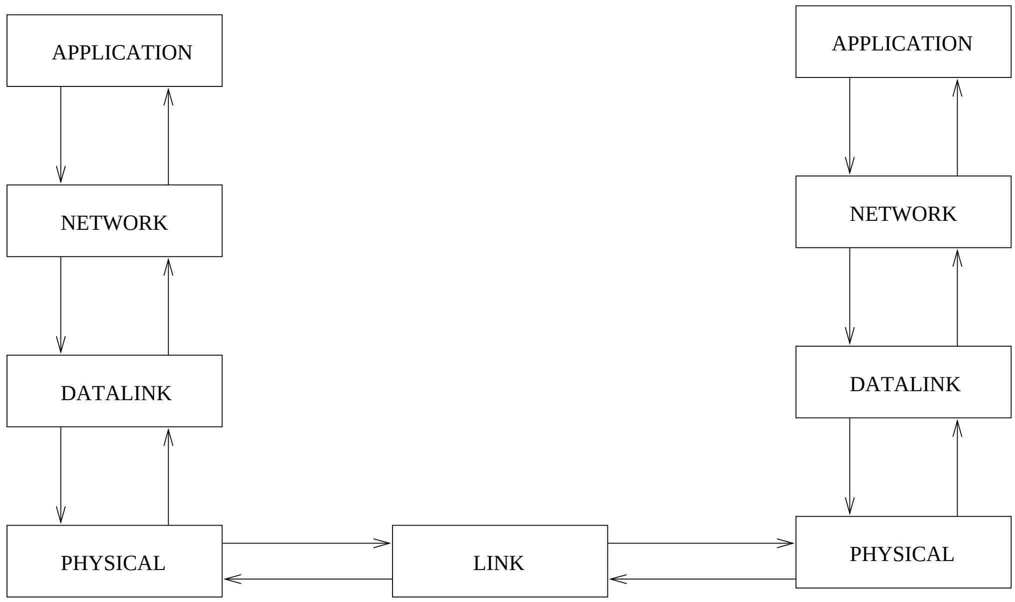

Figure 1: Layered Architecture

2 Layered architecture

For this lab, assume that each node in the network has four layers: *physical layer*, *datalink layer (DLC)*, *network layer* and *application layer*. Nodes in the network are connected to one another via *links*. Each layer in a node can be thought of as an abstract *entity* that performs certain functions. Similarly, links are also entities that have some functionality.

Figure 1 outlines the four layers in a node connected by a link entity. **In this lab, you will implement the fragmentation and reassembly algorithms of a simple network layer.**

3 Protocol Data Units

Each layer communicates through Protocol Data Units (PDU). The application layer PDU is called **A_PDU**, the network layer PDU is called **N_PDU**, the DLC PDU is called **D_PDU**, and the physical layer PDU is called **PH_PDU**. The pdu formats are defined at the end of this section. These definitions, together with some others are provided to you in the file `pdu.h` .

Communication is performed as follows: The application layer sends an `a_pdu` to the network layer. The network layer receives this `a_pdu` and encapsulates it within a `n_pdu`. It then performs its functions on the `n_pdu` and sends the `n_pdu` to the datalink layer. In the same manner, the datalink layer receives the `n_pdu`, encapsulates it within a `d_pdu` and sends it to the physical layer which in turn sends it to the link entity. The link entity receives a `ph_pdu` from one physical layer and delivers it to the physical layer at the other end. When a physical layer receives a `ph_pdu` from the link, it extracts the `d_pdu` from it and sends it to the dlc layer which in turn sends the `n_pdu` within the `d_pdu` to the network layer. The network layer at the destination extracts the `a_pdu` and sends it to the application layer after performing reassembly functions. The network layer at a router, however, has no application layer. It may fragment a packet into several fragments depending upon the maximum transmission unit (**MTU**) size of the next subnet hop and send the fragments to the appropriate data link layer.

In our implementation, the application service data unit size is 1024 bytes and the a_pdu size is 1032 bytes. The a_pdu is encapsulated in an n_pdu which is defined similar to an IP packet. We bypass the implementation of variable size packets by using subsets of a large data field. The other parts of the header are exactly the same as in IP. In particular, we do not implement IP options. Hence the header length is always 20 bytes (IPMHLEN). You cannot add or delete fields from the header. In some cases you might need to do bit-wise operations (eg. for the MF flag and the 13-bit fragment offset). Also note that the ip_len field in the IP header is the length of the payload plus the header.

The other data structures include reassembly data structures (a fragment list data structure) and a structure for a subnet connection, called a 'dlc port.' The latter is not required for your programming and is not discussed. The `NETWORK_LAYER_ENTITY_TYPE` definition elsewhere includes a list of subnet connections and a fragment list table.

```
#define DATASIZE  1024
typedef struct {
     int  snode;           /* source node address      */
     int  dnode;           /* destination node address */
     char data[DATASIZE];  /* message                  */
} A_PDU_TYPE;
#define A_PDU_SIZE sizeof(A_PDU_TYPE)
```

```
#define MAX_IP_PAYLOAD  2048 /* artificial: to avoid impl. var size pkts */
typedef int IPaddr;

typedef struct ip {
     u_char     ip_verlen;   /* IP version and header length: (ignore) */
     u_char     ip_tos;      /* type of service:  (ignore) */
     short      ip_len;      /* total packet length (in octets): USED */
     short      ip_id;       /* datagram id  : USED */
     short      ip_fragoff;  /* fragment offset (in 8 octets)+ flags: USED */
     u_char     ip_ttl;      /* time to live in gateway hops: (ignore)  */
     u_char     ip_proto;    /* IP protocol: (ignore) */
     short      ip_chksum;   /* header checksum: (ignore) */
     IPaddr     ip_src;      /* IP address of source: USED */
     IPaddr     ip_dst;      /* IP address of destination: USED */
     u_char     ip_data[MAX_IP_PAYLOAD];  /* variable length data: USED */
} N_PDU_TYPE;
#define N_PDU_SIZE sizeof(N_PDU_TYPE)

#define IP_MF 0x2000      /* more fragments bit mask (8192 in base 10) */
#define IP_DF 0x4000      /* don't fragment bit: (ignore) */
#define IP_FRAGOFF 0x1fff /* frag offset mask:
                             masks out the top-3 bits of a 16-bit `short' */

#define IPMHLEN 20   /* min (standard) IP header length (in bytes) */


#define IP_FQSIZE 100  /* Max number of frag queues */
#define IP_MAXNF 100  /* Max number of frags/datagram (ignore) */
#define IP_FTTL 6    /* Max time between 2 frags (6 ms) */
#define FTTL_DECR 1    /* 1 ms decrement of FTTL  (ignore) */
#define FTTL_DECR_USECS 1000 /* = 1000 usecs decrement of FTTL (ignore) */

/* ipf_state flags */
#define IPFF_VALID  1   /* contents are valid */
#define IPFF_BOGUS  2   /* drop frags that match */
#define IPFF_FREE  3   /* this queue is free to be allocated  */

/* Fragment queue and associated state variables */
/* The destination network layer will have a table of these structures */
struct ipfq {
     char         ipf_state;      /* VALID, FREE or BOGUS  */
     IPaddr            ipf_src;        /* IP address of the source */
     short             ipf_id;         /* datagram id */
     int               ipf_ttl;        /* countdown to disposal */
     int               nfrags;         /* variable to aid debugging */
     OrderedList_t     *ipf_q;         /* ordered list of fragments   */
};


/* Fragment list table and dlc ports in NETWORK_LAYER_ENTITY_TYPE
   definition (not in pdu.h) */

typedef struct {
```

```
      .
      .
      .
 DLCPort_t      dlcport[MAXPORT]; /* dlc port variables (not*/
 struct ipfq    ipfqt[IP_FQSIZE];/* Fragment list table  */
 } NETWORK_LAYER_ENTITY_TYPE;
```

4 Service Primitives

Inter-layer communication takes place by means of *service primitives.* The network layer uses functions called `send_pdu_to_datalink()` and `send_pdu_to_application()` to send pdu to the datalink and application layers respectively. The only difference is that `send_pdu_to_datalink()` has an additional argument specifying the index of the data link to which the packet is to be sent. In this lab, you will design the fragmentation and reassembly algorithms of a simple network layer.

5 Fragmentation and Reassembly

This section gives a short tutorial on fragmentation and reassembly procedures in IP. Some details of the procedures are also described along with the big picture of the distributed algorithm. The IP header is shown for reference in figure 2 and the data structure definition (`N_PDU_TYPE`) is given in the previous section.

The IP protocol standard specifies that all implementations of IP must be able to fragment and reassemble datagrams. In practice, any router that connects two or more networks with different MTU sizes must fragment datagrams. In our implementation, we assume that the source does not fragment packets, i.e., the packet size is smaller than the MTU of the first hop. However, routers may need to fragment these packets.

5.1 Fragmenting datagrams

Fragmentation occurs after IP has routed a datagram (i.e., decided the next hop subnet) and is about to deposit it on the queue associated with a given network interface (i.e., the data link connection). IP compares the datagram length to the network MTU to determinate whether fragmentation is needed. In the simplest case, the entire datagram fits in a single network packet of frame, and will not need fragmentation. Otherwise, fragmentation is needed. If the "Do not Fragment" (DF) bit were set, the original packet should be dropped and an ICMP packet should be sent to the host. No fragments are created in this case. However, for this lab, we assume that this bit is never set, and can be ignored. The DF bit is zero when the ip_fragoff field is initialized to zero.

In such cases, IP creates multiple datagrams, each with the fragment bit set, and places consecutive pieces of data from the original datagram in them. It is the data portion of the original packet which is fragmented. Each fragment has a header separately constructed

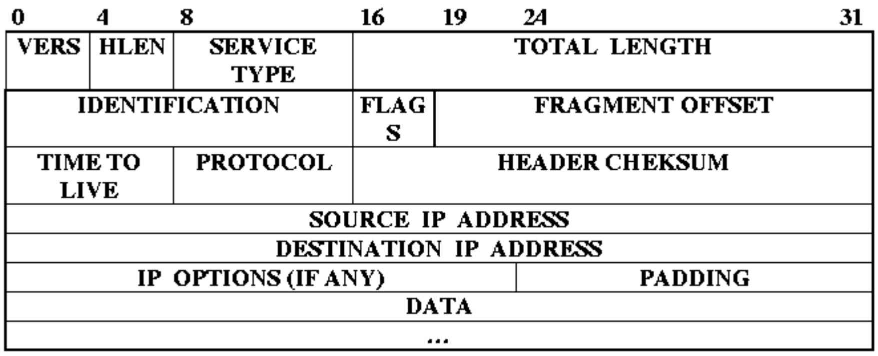

Figure 2: IP Header

which bears strong resemblance to the original packet header, but has the fragmentation fields set. Specifically, it sets the "more fragments" (MF) bit in the IP header of all fragments from a datagram except for the fragment that carries the final octets of data. As it constructs fragments, IP passes them to the network interface for transmission.

5.2 Fragmenting Fragments

Fragmentation becomes slightly more complex if the datagram being fragmented is already a fragment. Such cases arise when a datagram passes through two or more routers. If one router fragments the original datagram, the fragment themselves may be too large for a subsequent network along the path. Thus, a router may receive fragments that it must fragment into even smaller pieces.

The subtle distinction between datagram fragmentation and fragment fragmentation arises from the way a router must handle the "more fragments" (MF) bit. When a router fragments an original (unfragmented) datagram, it sets the MF bit on all but the final fragment. Similarly, if the MF bit is not set on a fragment, the router treats it exactly like an original datagram and sets the MF bit in every subfragment except the last. When a router fragments a non-final fragment, it sets the MF bit on all (sub-)fragments it produces because none of them can be the final fragment for the entire datagram.

The function `FragmentPacketAndSendToDlc()` makes the decision about fragmentation and sends the fragments. If the packet length is less then the network MTU, the datagram is sent calling `send_pdu_to_datalink()`. If the datagram cannot be sent in one packet it is divided in into a sequence of maximum possible fragment length, which must be a multiple of 8, plus a final fragment of whatever remains.

To do so, first the maximum fragment size (`maxdlen`) is computed, then the program iterates through the datagram, calling `send_pdu_to_datalink()` to send each fragment. The maximum amount of data that can be sent equals the MTU minus the IP header length, *truncated to the nearest multiple of 8*. You can obtain this by clearing the last three bits (`& ~7` in C-code). The iteration proceeds only while the data remaining in the datagram is strictly greater than the maximum that can be sent. The iteration will stop before sending the last fragment, even in the case where all fragments happen to be of equal size. When sending the final fragment, the MF bit is not usually set. But, in the case where a router happens to further fragment a non-final fragment, it must leave the MF bit set in all fragments.

5.3 Datagram Reassembly

Reassembly requires IP on the receiving machine to accumulate incoming fragments until a complete datagram can be reassembled. Once reassembled, IP routes the datagram on toward its destination. Because IP does not guarantee order of delivery, the protocol requires IP to accept fragments that are delayed or arrive out-of-order. Furthermore, fragments for a given datagram may arrive intermixed with fragments from other datagrams.

To make the implementation efficient, the data structure used to store fragments must permit: quick location of the group of fragments that comprise a given datagram, fast insertion of a new fragment into a group, efficient test of whether a complete datagram has arrived, timeout of fragments, and eventual removal of fragments if the timer expires before reassembly can be completed.

Reassembly software must test whether all fragments have arrived for a given datagram. To make the test efficient, each fragment is stored in a list. In particular, the fragments on a given list are ordered by their 13-bit fragment offset from the original datagram. The IP protocol design makes the choice of sort key (the 13-bit offset) easy because even fragmented fragments have offsets measured from the original datagram. Thus, it is possible to insert any fragment in the list without knowing whether it resulted from a single fragmentation or multiple fragmentation. In our implementation `ipreass()` is the reassembly subroutine. It calls `ipfadd()` that add a fragment to a fragment list. When all fragments are arrived, `ipreass()` calls `ipfjoin()` to join fragments, which in turn calls `ipfcons()` to reconstruct the datagram. There is another routine `ipftimer()` which is called periodically (once every 1 ms in our implementation and once every 500 ms in reality). This routine updates the time-to-live variables in the currently valid fragment list table entries and may clean up lists for which the time-to-live variable has decreased to zero. In our implementation, we ignore the functionality of the 'BOGUS' state of the fragment table entry. In real networks, the table entry may be marked bogus for a long time and removed only after a timeout to avoid getting late fragments.

5.4 Adding a fragment to a queue

The IP protocol uses information in the header of an incoming fragment to identify the appropriate list. Fragments belong to the same datagram if they have identical values in *both* their source address and IP identification fields. Subroutine `ipreass` takes a fragment, finds the appropriate list and add the fragment to the fragment list. Given a fragment it searches the fragment array to see if it contains an existing entry for the datagram to which the fragment belongs. At each entry, it compares the source and identification fields, and calls `ipfadd()` to add the fragment to the list if it finds a match. It then calls `ipfjoin()` to see if all fragments can be reassembled into a datagram. If no match is found in the fragment table, `ipreass()` allocates the first unused entry in the array, copies in the source and identification fields, and places the fragment on a newly allocated queue.

5.5 Testing for a complete datagram

When adding a new fragment to a list, IP must check to see if it has all the fragments that comprise a datagram. Procedure `ipfjoin` examines a list of fragments to see if they form a complete datagram. After verifying that the specified fragment list is in use, `ipfjoin` enters a loop that iterates through the fragments. It starts a running variable at zero, and uses it to see if the current fragment occurs at the expected location in the datagram. First, `ipfjoin` checks to see that the offset in the current fragment matches the running variable. If the offset of the current fragment exceeds the running variable, there must be a missing fragment and `ipfjoin` returns zero (which means that the fragment cannot be joined). If the fragment matches, `ipfjoin` computes the expected offset of the next fragment by adding the current fragment length to the running variable. Finally after reached the end of the fragment list, `ipfjoin` calls `ipfcons` to collect the fragments and rebuild a complete N_PDU.

5.6 Building a datagram from fragments

Procedure `ipfcons` reassembles fragments into a complete N_PDU. In addition to copying the data from each fragment into place, it builds a valid datagram header. Information for the datagram header comes from the header in the first fragment, modified to reflect the full datagram's size. The procedure then sets the fragment offset field to zero and releases the buffers that hold individual fragments, and the state variables' entry in the fragment list table `ipfqt`.

5.7 Timeout for old fragment lists

Due to errors in the network, some fragments may not reach the destination. In such a case the partial fragment lists corresponding to these packets must be eventually removed. IP uses a time-to-live variable in the `ipfqt` entry which is re-initialized to a default value (we use 6 ms, real implementations use 60 sec) everytime a new fragment is received. Periodically, a timer routine (`ipftimer()`) is called which decrements this value by a constant amount (1 ms in our implementation, 500 ms or 1 s in reality) for all valid table entries. When the variable reaches zero, the table entry is freed, and the corresponding fragments are dropped. Real implementations send an ICMP message to the source, but we do not send ICMP messages.

6 Function descriptions

The following functions are given in network_layer.c. You have to fill in the code for the functions indicated. You should study each of the functions carefully to understand what they are expected to do.

- `NetworkLayerReceive(NETWORK_LAYER_ENTITY_TYPE *network_layer_entity, GENERIC_LAYER_ENTITY_TYPE *generic_layer_entity, PDU_TYPE *pdu)`: Processes a received packet.

  You should NOT modify this function.

- `NetworkToDatalink(NETWORK_LAYER_ENTITY_TYPE *network_layer_entity, PDU_TYPE *input_pdu)`: A source or router can call this routine. It chooses an output datalink (routing) and calls fragmentation subroutines.

  You should NOT modify this function.

- `NetworkToApplication(NETWORK_LAYER_ENTITY_TYPE *network_layer_entity, PDU_TYPE *pdu_from_datalink)`: Called by the destination node. It reassembles packet if necessary. If reassembly is complete, it forms an `a_pdu` and sends it to the application.

  You should NOT modify this function.

- `FragmentPacketAndSendToDLC(NETWORK_LAYER_ENTITY_TYPE *network_layer_entity, PDU_TYPE *input_pdu, int i)`: Called by a router with a target subnet or datalink (indexed by 'i') specified as an argument. Fragments packets if necessary according to the IP protocol. Special handling for fragments of fragments and the last fragment.

  You need to write this function.

- `ipreass(NETWORK_LAYER_ENTITY_TYPE *network_layer_entity, PDU_TYPE *pdu)`: Reassembles packet if necessary. If 'pdu' is a fragment, it calls `ipfadd()` to add it to a fragment list in the fragment list table at the destination. A new fragment list is created if there is no match in the fragment list table. If the reassembly is complete, a reassembled n_pdu is constructed (calls `ipfjoin()` which in turn calls `ipfcons()`). Returns NULL if reassembly incomplete.

  You need to write this function.

- `ipfadd(NETWORK_LAYER_ENTITY_TYPE *network_layer_entity, struct ipfq *iq, PDU_TYPE *pdu)`: Adds 'pdu' to an ordered fragment list (indexed by the 13-bit fragment offset) in 'iq' and updates state variables.

  You need to write this function.

- `ipfjoin(struct ipfq *iq)`: Traverses the ordered fragment list in 'iq' to see if all required fragments have arrived. If so, it calls `ipfcons()` to construct a reassembled n_pdu. Else, it returns NULL.

  You need to write this function.

- `ipfcons(struct ipfq *iq)`: Dequeues fragments from the ordered fragment list in 'iq' and constructs a reassembled pdu out of them. The ordered list structures and state variables in 'iq' are released. Returns the reassembled pdu.

  You need to write this function.

The following functions are provided to you. You DO NOT NEED TO KNOW the implementation details of these functions. You only need to know the functionality as described below.

- `send_pdu_to_datalink(NETWORK_LAYER_ENTITY_TYPE *network_layer_entity, PDU_TYPE *pdu, int i)`: Sends the (n_)pdu to the datalink layer of subnet 'i'.

- `send_pdu_to_application(NETWORK_LAYER_ENTITY_TYPE *network_layer_entity, PDU_TYPE *pdu)`: Sends the (a_)pdu to the application layer.

- `DropPDU(PDU_TYPE *pdu)`: Frees the PDU.

- `GetMTU(NETWORK_LAYER_ENTITY_TYPE *network_layer_entity, int i)`: Returns the MTU of the subnet (or dlc) 'i'.

- `choose_output_datalink(NETWORK_LAYER_ENTITY_TYPE *cn)`: Does the routing function. Chooses an output datalink (subnet) for the packet. The choice is not random (to allow easy debugging), but all available output datalinks are chosen over a period of time.

- `FormPacketsAndSendToDLC(NETWORK_LAYER_ENTITY_TYPE *network_layer_entity, PDU_TYPE *input_pdu, int i)`: Source generates packets with unique ids.

- `CreatePacket(int src, int dest, short pktlen, short pktid, short fragoff, char *sourcep)`: Creates a n_pdu given the source address 'src', destination address 'dest', packet length (including header) 'pktlen', unique packet id 'pktid', 13-bit fragment offset 'OR'ed with 3 most-significant bits of flags (IP_MF or IP_DF), 'fragoff', and the pointer to source data.

- `ipftimer(NETWORK_LAYER_ENTITY_TYPE *network_layer_entity)`: Called periodically. Updates time-to-live fields and deletes expired fragment lists in the fragment list table at the destination.

- `CreateOrderedList()`: Creates an ordered list and returns a pointer to a value of type OrderedList_t.

- `AddToOrderedList(OrderedList_t *q, PDU_TYPE *pdu, int fragoff)`: Adds pdu to ordered list based upon the 13-bit ordering key 'fragoff'. The 3 most-significant bits of 'fragoff' are zero.

- `TraverseOrderedList(OrderedList_t *q, int *position_ptr)`: Returns the pdu in the ordered list 'q' at position indicated by 'position_ptr' and increments '*position_ptr'. If '*position_ptr' is zero, then the first element of the ordered list is returned. Illegal values of 'position_ptr' result in program termination. Successive calls to this routine starting from '*position_ptr' value of zero can be used to traverse the ordered list, 'q'. When the function returns NULL, you are at the end of the list.

- `DequeueFromOrderedList(OrderedList_t *q)`: Dequeues the next element in the ordered list and returns the pdu stored in it.

- `FreeOrderedList(OrderedList_t *q)`: Releases the resources of the ordered list 'q'. Must be called before using `CreateOrderedList()` with the same 'q'.

7 Files and Configurations

The lab is available in two versions - one that runs on campus IBM AIX and another that runs on Intel Linux machines. Both versions are the same. Since remote access can be hard and slow for off-campus students, we recommend that they use the Linux version. However, they will need to have a Linux machine at their end.

- **Downloading files for campus IBM AIX**:
  Since your disk space quota on rcs is limited, please follow these instructions carefully:
  - Create a unique directory in /tmp and copy the file `/home/81/kalyas/public/Lab1/Lab1.tar.gz`, into that unique directory.
  - `gunzip Lab1.tar.gz`
  - `tar xvf Lab1.tar`
  - It will create a directory called `''Files''`. Move that directory into your account file directories.

- **Downloading files for Intel Linux**:
  - Download the file Lab1-Linux.tar.gz from the course webpage.
  - `tar xzvf Lab1-Linux.tar.gz`
  - This creates a directory called "student_frag". Change directory to student_frag.

- The directory "Files" will contain the following files:
  - `pdu.h`: header file containing some data structure declarations and definitions. You don't need to include this file anywhere in your source code because it is already included in `network_layer.h`. You will need to use some of the function definitions provided in this file, like `pdu_alloc()` and `pdu_free()`
  - `network_layer.c`: file containing the outline for the lab. **You will have to DO ALL YOUR CODING ONLY IN THIS FILE**.

- `makefile`: makefile for the lab. Running `make` will create an executable called `Frag_exec`
- `Frag_demo`: a sample executable file to familiarize you with the graphical user interface. You need to do part 1 (see below) of the lab using this executable.
- Six configuration files: `*.config`. These files specify the configuration of the network. In this lab, we will use a "multi-router" configuration as shown in figure 3. The configuration is described in detail below.

8 The Multi-Router Configuration

The configuration consists of a single source and a single destination connected by a network of routers and unidirectional datalink (subnet) connections. Every data link connection is labeled either "input" or "output." Each network layer is either a "source", "destination" or a "router."

Routing is simple. A source or a router simply chooses any output data link to transmit the packet and the packet will reach the destination if there are no errors. Packets or fragments with error are dropped at the next data link layer on the path. The source or router may choose different output data links for different packets. Once a data link is chosen for a packet, the packet may be fragmented depending upon the MTU of the data link. Link lengths may be different and links may introduce errors in certain configurations.

Graphical interface support for the lab is provided in the following ways: packet lengths on links are shown proportional to their actual sizes. Data link and physical layers are not shown to avoid cluttering on the screen. Errored packets are shown to change color on a link. The destination has a separate window which shows the status of the reassembly, i.e., the set of packets being currently reassembled, and the lengths of fragments received.

The configuration files specify different parameters for the simulations:

frag1.config: Same MTU sizes, same link delays, no link errors.

frag2.config: Different MTU sizes, same link delays, no link errors.

frag3.config: Different MTU sizes, different link delays, no link errors.

frag4.config: Different MTU sizes, different link delays, link errors present.

frag5.config: Same MTU sizes, different link delays, link errors present.

The MTU sizes, link delay, and link error probability values for the configurations are as follows. When the MTU sizes are the same, the value is 2048 bytes (which is enough to carry the 1032 bytes N_PDU payload plus 20 bytes of IP header without fragmentation in the network). When the link delays are the same, the value is 5 microseconds each.

When the MTUs, link delays or error probabilities are different, the following values are used in the links (for link labels see figure 3) depending upon the configuration (decides which parameter is different):

Link 1 MTU = 1518 bytes, Link delay = 5 usecs, Error Probability = 0.005.

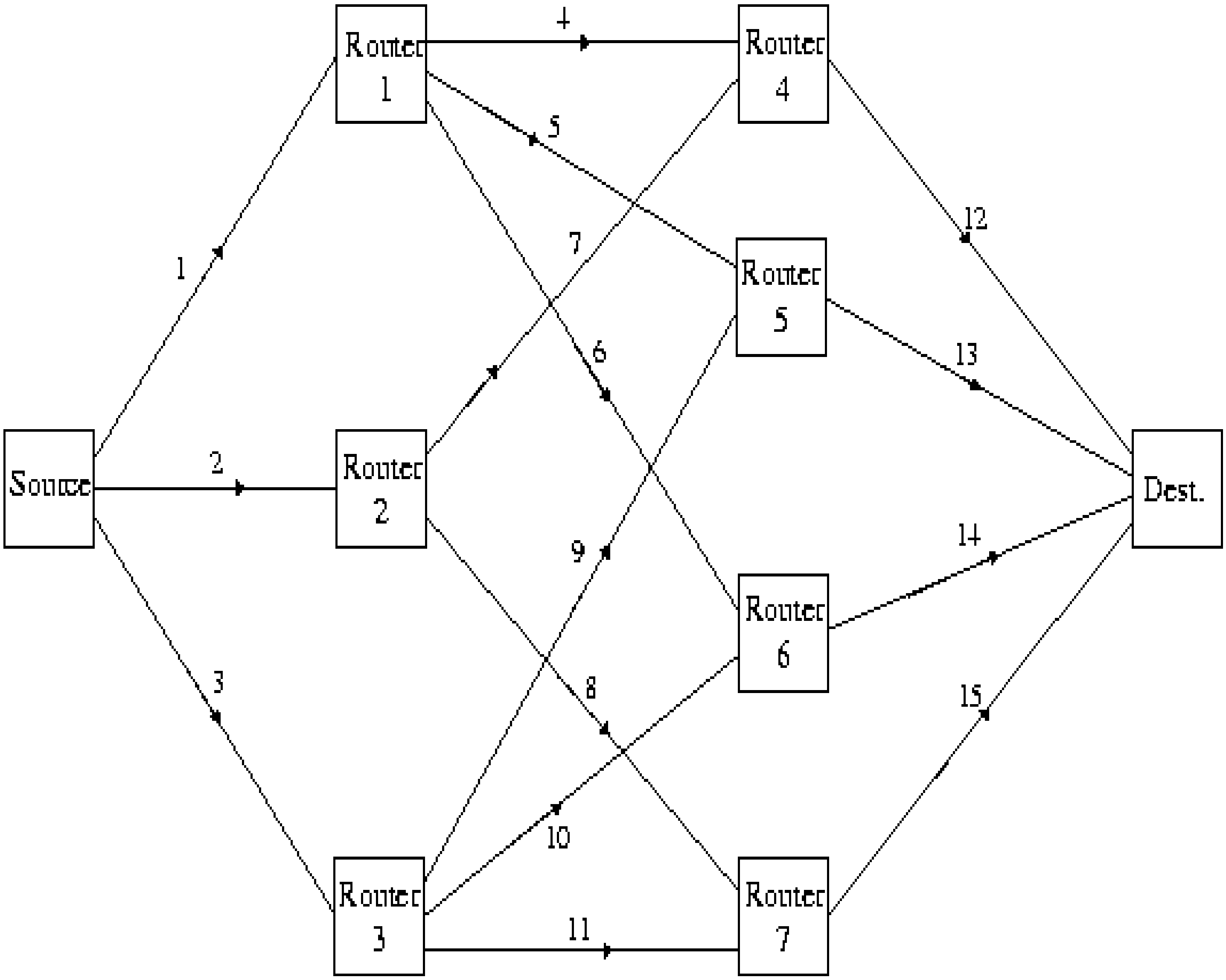

Figure 3: Multi-Router Configuration

Link 2 MTU = 1518 bytes, Link delay = 20 usecs, Error Probability = 0.005.

Link 3 MTU = 1518 bytes, Link delay = 50 usecs, Error Probability = 0.005.

Link 4 MTU = 576 bytes, Link delay = 20 usecs, Error Probability = 0.1.

Link 5 MTU = 256 bytes, Link delay = 100 usecs, Error Probability = 0.15.

Link 6 MTU = 784 bytes, Link delay = 5 usecs, Error Probability = 0.2.

Link 7 MTU = 512 bytes, Link delay = 25 usecs, Error Probability = 0.15.

Link 8 MTU = 256 bytes, Link delay = 30 usecs, Error Probability = 0.1.

Link 9 MTU = 128 bytes, Link delay = 200 usecs, Error Probability = 0.05.

Link 10 MTU = 1024 bytes, Link delay = 100 usecs, Error Probability = 0.1.

Link 11 MTU = 576 bytes, Link delay = 25 usecs, Error Probability = 0.15.

Link 12 MTU = 1024 bytes, Link delay = 50 usecs, Error Probability = 0.05.

Link 13 MTU = 576 bytes, Link delay = 10 usecs, Error Probability = 0.05.

Link 14 MTU = 256 bytes, Link delay = 25 usecs, Error Probability = 0.05.

Link 15 MTU = 192 bytes, Link delay = 100 usecs, Error Probability = 0.05.

In addition, the network layer has a 20 usec processing delay for packetization (or fragmentation) and reassembly of every fragment, and for sending an a_pdu to the application layer. Specifically, packets arriving from different data links at the destination are serialized, and a processing delay of 20 usec per packet is added.

The experiments you need to do with the above configurations are listed in the next section.

9 Running Your Program: The Graphical User Interface

9.1 Running your program on campus IBM AIX machines

- **On-Campus Students with access to campus IBM AIX machines**:
  Use your RCS id to login. You should be able to compile and run the graphical interface from a terminal window.

- **Off-campus students (or students remotely logging into a campus IBM AIX machines)**:
  Login to a campus IBM-AIX machine on rcs using your account. Please contact the help desk to get a partial list of IBM AIX machines on campus. Remember that you need X-windows/Unix capability on the remote side in order to see the GUI. You also need a high speed connection for zippy response. Also, your machine MUST NOT be behind a firewall (firewalls kill the GUI). You will do all your compile and running of

the programs by remotely logging into AIX machines on campus. You can edit and code using a local editor program and ftp over the code for compilation to rcs.

Make sure you export DISPLAY to your local UNIX machine. See the FAQ for questions regarding exporting DISPLAY.

9.2 Running your program on Linux Machines

Recommended for OFF-CAMPUS students.

- Download the tarfile to your machine and gunzip and untar it using
  tar xzvf Lab1-Linux.tar.gz
- This will create a directory called "student_frag". Change directory to student_frag.
  cd student_frag

9.3 General Notes on running your program

- In the`Files` directory, type `Frag_demo frag1.config`. In general, invoke the demo with a configuration file as follows: `Frag_demo <config name>`
- A window with the simulator interface will open on your terminal. You should also see two small windows to the lower left and right of the screen. The main window has a network diagram with a single source, a single destination and a bunch of routers. The source and destination nodes have two layers (network and application) denoted by the squares, while the routers have only one layer (network). The data link and physical layers have been hidden to avoid cluttering the screen. The network of links and routers should be clearly seen.
- In addition to the simulator window, there should be two smaller windows loaded up. For each application layer, there is a send/receive window where you can load a graphic ("earth") and send it to the other end. Once the transmission starts, a reassembly window will open up near the destination and show yellow bands with red borders denoting partial fragments of packets which have arrived. When a packet is reassembled, it is sent to the application which plots it on its screen.
- The toolbar on the top left of the main window contains a "Run" menu item which has a couple of useful options:
  - A debug level option where you can increase the debug level to see some debugging text output in the terminal.
  - A delay option which can be set to 1 (to speed up the simulation)

  In brief, the steps to operate the demo are:
  - Start the demo by loading a configuration: `Frag_demo <config name>`
  - Set debug level option (from the "run" menu options) to 1.
  - Set the delay option (again from the "run" menu options) to 1.

- Load the picture (graph) of ”earth” in the LEFT (source) application window.
- Click the send button in the LEFT (source) application window to send the ”earth” picture to the right application window. DO NOT load or send the “earth” picture from the right application window (app2).
- Another window called the “reassembly window” will pop up. To observe it closely, you can pause the simulation and scroll the screen. The “earth” screen generates has 328 a_pdus. Sit back and watch the action.

In your work for the deliverables, you can use the debug option described below.

- **Debugging note:** The code contains couple of accounting variables to count the number of packets and the number of fragments received at the destination, and the following dprintf statement:

```
if (TICKS_TO_USECS(ev_now())/1000.0 > 17.0){
     dprintf(1,"Time: %8.3f ms. Node %s. Received %d pkts, %d fragments\n",
     TICKS_TO_USECS(ev_now())/1000.0, network_layer_entity->cn_name,
     network_layer_entity->num_packets,
     network_layer_entity->num_fragments);
     fflush(stdout);
}
```

To use this, you need to increase the debug level once (from the ”run” menu options). Specifically:

Once the simulation comes close to termination, you will see a dump of information on your screen. You can just choose the latest time, packets, fragments information for your purposes. But you will need to interpret the results, especially in the error configurations.

You can use the above style of dprintfs for your statistics in part 2 (see below) as well. If you need to define any variables, you can copy the file network_layer.h from: `Files/defaults/components.src/` to the directory `Files/` and add your variables (just like num_packets and num_fragments).

- **Additional Debugging Note:** In the code, ev_now() gives the current simulation time in simulator ticks. the macro TICKS_TO_USECS() converts a quantity from ticks to microseconds, and the variable network_layer_entity-¿cn_name outputs the name of the network layer as a character string. You can use this in part 2 below.

10 Miscellaneous Notes

- For questions, FIRST CHECK THE FAQ (on course web page), and then send post questions to the bulletin board ONLY. **Do not** send emails directly to the instructor or TAs. The TAs are responsible for prompt replies to your technical questions on the bulletin board. **Also, the TAs will NOT DEBUG your code.**

- TA office hours:
  Ye Tao: Fri 3-4pm at JEC 6212, x8289
  Kartik: Tue 2-3pm at JEC 6212, x8289
  Satish: Wed 2-3 pm at JEC 6212, x8289
  Adnan El-nasan: Mon 2-3pm, Thr 2-3pm at JEC 6031

- **Note for ON-Campus students regarding AIX machines on campus**: The following are the only IBM AIX machines on campus: rcs-ibm1.rpi.edu, rcs-ibm2.rpi.edu. "rcs-ibm1" is a J40 with 4 processors and 512Mb of memory and rcs-ibm2 is a J50 with 6 processors and 512Mb of memory. Both machines are robust and currently have a total of 74 logins at this time, with 12am-8am seeing the lowest use. During peak usage, normally between 10am and 6pm, it is not uncommon to see a couple hundred logins. The machines are both configured to prevent single users from significantly impacting available CPU time, by killing processes that single-handedly consume more than an hour of CPU time. Between this and automatic load balancing between the processor, the machines do not bog down easily. Currently the load average for the last 15 minutes has been 16% and 11% on the two machines, and each can reach up to 200-300% per processor before significantly impacting user speeds, particularly for quick processes. Please use this information as you plan to access these machines.

- **Note for Linux Version of the lab**: When the Lab1 is run, the color of the output (Earth) might be BLUE. This is perfectly allright, DO NOT PANIC.

11 Submissions

The submissions are to be made through the drop-boxes in WebCT. You need to submit a copy of your code, i.e., *network_layer.c* and the answers to the questions asked below in part 1, 2, 3 and 4. Please submit the answers to these questions in *text* format and name your file as *solution.txt*.

- **Part 1**: The questions below are intended to clarify the concepts of fragmentation and are intended to be a primer for the coding exercise. The variable names used in the problems are same as those needed for coding.

  An application sends a packet of 1960 bytes to the TCP, which adds its 20 bytes of header of its own and passes the packet to IP. Assume that the IP header is of 20 bytes and the fragment offset is 0.

  1. Now if the path MTU has been found to be 576 bytes then calculate the following:
     - Number of fragments
     - IP Payload
  2. If the IP Payload is called tosend, then calculate the following values for each fragment
     - Frag_off, where frag_off is the fragment's offset.
     - Dataoffset, where dataoffset is the offset of the first data byte from the original payload's start (of data).

– Update the value of tosend, to reflect the bytes remaining to be sent.

Do the same problem if fragment offset in the IP Header is 0xA002.

- **Part 2**: Answer the following questions using **Frag_demo**:

  1. In each simulation run (with the five configurations: frag1.config through frag5.config) **what is the average number of fragments per packet ?** Eg: In frag1.config, a total of 328 packets and 328 fragments are received. The average in this case is 1.
  2. In the error configurations, the quality of the received picture is different in different cases. Get a window capture of the "earth" received and use the reassembly window data (number of packets dropped) to **qualitatively and quantitatively** assess the performance in each case. Especially for frag4.config and frag5.config.

- **Part 3**: Implement the fragmentation and reassembly procedures as described in the earlier sections.

  – Study the source code files carefully. (don't worry about the config files).
  – Now you are ready to write your program for the network layer. Do all your coding in network_layer.c - the comments and the prior sections should guide you in this process.
  – You can progressively test your implementation with the various configurations starting with frag1.config (no error, equal link delays, no fragmentation). You can uncomment and use the suggested dprintf statements to help you guide debugging. The reassembly window at the destination should also help you do the debugging. There are a few lines of graphics support in the code. Do not remove them. You will not see the reassembly window correctly if you do. To compile your program, type **make**. This will produce an executable called `Frag_exec` in your working directory.

The deliverables in this parts are as follows.

1. Execute your version of the code and use the configuration files to make sure it works.
2. Calculate the total number of packets received, total number of fragments and the average number of fragments per packet in frag2.config. [effect of MTU sizes]
3. Calculate the maximum number of partially reassembled fragments in frag3.config. This is also the minimum size of the retransmission table. [effect of link delays with fragmentation].
4. Calculate the packets dropped (due to timeout or loss of all fragments) and number of fragments lost in the network in frag4.config [effect of errors with fragmentation].

   Compare this with the number of packets dropped (one fragment/packet) in frag5.config.

You can use information from the previous configurations about the total number of packets and total number of fragments expected in this configuration.

5. You need to **copy of your code (network_layer.c)** and the list of statistics mentioned above. The code should well-commented and formatted (indented) to facilitate easy grading. There are no graphs to submit in this lab.

- **Part 4**: A **Short Summary** of the lab.