Reliable Data Transmission
CS5516: Project 1
Spring 2000

Introduction
Reliable data transmission protocols form the basis of data communication. The application interface to the network assumes that the network is capable of transmitting and receiving their data without introducing any errors. In contrast, the network layer protocols run on top of data link layers are often noisy, and depending on their technology - for instance modems - they are prone to significant data loss.

In this project, you will implement a reliable data transmission protocol that runs on top of an unreliable data link layer. The network setup consists of two nodes - node 0, the transmitter and node 1 the receiver. The nodes are directly connected to each other over a lossy point-to-point cable. Some of the messages transmitted will not be received. Your reliable transmission protocol should ensure that messages sent by the transmitter are successfully received at the receiver. For the purposes of this project, you may implement any of the sliding window protocols (go back N, selective acknowledgement or negative acknowledgement) discussed in class, or any other equivalent scheme. As a last resort, you may implement the stop and wait protocol for lower credit.

Simulator
Your implementation will run over a simulator, which models a functional interface to a two node 10Mbps Ethernet network with a round trip propagation delay of 50us. Your code should interface with the simulator and add necessary functionality to ensure reliable transmission of messages.

To interface with the simulator, you need to provide C functions that implement reliable transmission of messages. The flow of control within the transmitter and the receiver is as follows:

Transmitter
1. Get the next message from the sending application
   a. This operation is performed by calling a function within the simulator called get_message()
   b. If get_message() returns NULL, all messages have been transmitted
2. Transmit the message and if set a timer for an acknowledgement.
   a. The transmit operation is performed by the write_msg() function within the simulator
   b. The simulator also provides access to a timer mechanism. Timers can be added by calling add_timer() and deleted by calling del_timer() with the appropriate parameters.
3. Repeat the above steps till all messages are transmitted successfully
4. End the simulation
   a. Call the end_simulation() function
Receiver

1. Receive the next message
   a. The simulator calls a function called `read_msg()`, with the contents of the packet and its length
2. Update internal state. Send an acknowledgement if necessary
3. Deliver the message to the receiving application
   a. This operation is performed by calling the `deliver_msg()` function
4. Repeat the above steps till all messages have been received and acknowledged
5. The simulation will terminate.
   a. NOTE: Only the transmitter should call the termination function. Why??

The simulator is implemented as a single-threaded program. Non-technically, this means that, both the transmitter and the receiver are implemented within the same piece of code running on one machine. In a real implementation, the transmitting and receiving code would run on two independent machines.

How does this affect your code? First, your code needs a mechanism to determine if it is running as the transmitter or the receiver and to take the appropriate action. This is achieved by calling `get_node_id()`, which returns an integer. If the result is 0, the code is running on the transmitter, if it returns 1, it is running on the receiver. Secondly, any global variables used by your code should be declared as arrays and accessed by using the result from `get_node_id()` as the index. For instance, instead of the following global declaration:

```c
int timeout;
```

In the simulator change it to:

```c
int timeout[MAX_NODES];
```

To use the global variable timeout, you need to reference it as `timeout[get_node_id()]`. This ensures that each node accesses its copy of the timeout variable instead of sharing the same global variable.

**Implementation Notes**

The simulator is available for download from the course web page as a library for Linux and FreeBSD. The Windows platforms are not and will not be supported. The simulator and the template source files are archived as a .tar file. Use the tar command as follows, to retrieve the contents of the archive.

```bash
tar cvf filename.tar
```

where `filename.tar` is either `project1.linux.tar` or `project1.bsd.tar`

You need to modify the file called `net_sim.c` to add your code. The file contains comments indicating the sections you need to modify. Compile the code by calling `make` and execute it by calling `./net_sim`.

You can write your entire code within `net_sim.c`. However, if you wish to create more source files, modify the `makefile` appropriately. This comment does not apply to most of you.
Results
At the end of the simulation, the simulator prints out several lines of statistics, such as total time taken, number of messages transmitted, number of messages lost etc. The simulator also indicates whether the transmitted message was received successfully at the receiver.

You need to submit an electronic version of your project by anonymous FTP to:

psmith.cs.vt.edu/pub/spring.2000/cs5516/upload

The submission should contain
- Source code: Modified net_sim.c and all header files
- All relevant binaries
- Output from the simulation run

Your source code should also contain a comment on the first few lines with the authors of the code, VT ID number, and the OS used to compile/test the code.

The submission format is tar. Use the tar program to archive the contents of your project directory as follows:
If your project directory is called project1 and it is under your home directory, then go to your home directory and issue the following command:

tar cvf filename.tar project1

filename.tar now contains the archived version of your project directory.

To submit the project, rename the archive (filename.tar) to <your ID>.<try number>.tar and upload the file.

<yör ID> is the ID number of one of the team members in the project.
<try number> is an integer between 1 and 3 (inclusive) that represents your submission number.

Each group is allowed 3 submissions with try numbers numbered 1, 2 and 3. Any submissions past 3 will be ignored. The last submission will be graded.

Simulator Functions
The functions offered by the simulator can be divided into three basic categories
1. Transmit/Receive Functions: Use these functions to transmit and receive messages
2. Timer functions: These functions provide access to high resolution timers. You can create as many timers as necessary
3. Utility functions: These functions provide an interface to basic facilities needed to implement a data link layer over the simulator.

This section presents an overview of the functions that comprise each category.
Transmit/Receive functions

void write_msg(unsigned short port_id, void *data, unsigned short len);

Input Parameters:
• port_id: Ignore this parameter for now. Always set it to 0
• data: pointer to the data to transmit
• len: Length of the data to transmit

Returns: Nothing

This function should be called to transmit a message or an acknowledgement. The function is
called with a pointer to the data to transmit and the length of the data.

void read_msg(unsigned short node_id, unsigned short from, unsigned short dev_id, void *data);

Input Parameters:
• node_id: Node identifier of the current node. This is either 0 or 1
• from: Node identifier of the transmitting node.
• dev_id: Ignore this parameter for now. It is always 0
• data: Pointer to the data

Returns: Nothing

This function should be defined by the user. It is called when the simulator receives data at a
node. For this project, the outline of the function has been defined in net_sim.c
Note: The function does not return the length of the data pointer. It is up to you to insert an
appropriate header to the message that contains a length field.

unsigned char *get_message(int *len);

Input Parameters:
• len: The length of the message is returned in this variable

Returns:
• A pointer to the data message. If there is no message it returns NULL

This function should only be called by the transmitter. It is used to get messages from the sending
application. The function returns the length of the message and a data pointer to the message.
The function should be called repeatedly to retrieve the entire message stream. It returns NULL
when all messages have been retrieved.
Note: For the purposes of this project, each message is composed of ASCII characters as a C
string. You may print the contents of the data pointer (as a string) returned by this function.

void deliver_message(unsigned char *data);

Input Parameters:
• data: Pointer to the data message

Returns: Nothing

This function should only be called by the receiver. It is used to deliver messages to the receiving application.
Note: This function should be given the same data that was sent by the sending application. If your code adds a header, you need to strip the header before calling this function.

**Timer Functions**

The timer functions make use of a structure to add and delete timers. The name and contents of the structure are as follows:

```c
typedef struct
{
    unsigned long data;
    unsigned int expires;
    void (*handler)(unsigned short, unsigned long);
} timer_event_type;
```

- **handler:** Pointer to a callback function that takes two parameters. The callback function will be called when the timer expires. It is commonly used to set a timeout function. The first parameter passed to the callback function is the current node identifier (0 or 1). The second parameter contains any data that you wish to pass to the callback function.
- **expires:** Time at which the callback function is called. It is usually set to current time + some timeout value.
- **data:** Any data that you wish to pass to the timeout function. It is commonly used to pass the sequence number of the packet to retransmit. Note: data is of type unsigned long. To pass a pointer value in this variable, you need to typecast it to an unsigned long.

```c
void add_timer(timer_event_type *tevent);
```

Input Parameters:
- **tevent:** Pointer to a timer_event_type structure

Returns: Nothing

This function adds a new timer. The appropriate callback function is called when the timer expires. Note: If you want to multiple simultaneous timers, you need to pass a different tevent structure to each call to add_timer. Reusing the tevent struct will erase previous timeouts. You may reuse the tevent struct after deleting the timer, or after the timer has expired.

```c
void del_timer(timer_event_type *tevent);
```

Input Parameters:
- **tevent:** Pointer to a timer_event_type structure

Returns: Nothing
This function deletes an existing timer specified by the `tevent` struct. If no such timer exists, the function returns without doing anything.

**Utility Functions**

```c
unsigned int get_current_time();
```

**Input Parameters:** None

**Returns:**
- The current time in clock ticks.

This function returns the current time in clock ticks elapsed since the simulator was started. Each clock tick represents 1us.

```c
unsigned int get_current_node_id();
```

**Input Parameters:** None

**Returns:**
- The node identifier of the currently executing node. It returns 0 for the transmitter and 1 for the receiver.

```c
void end_simulation();
```

**Input Parameters:** None

**Returns:** Nothing

This function should be called by the transmitter to end the simulation. The function should be called after all messages have been transmitted and acknowledged. The simulator prints some statistics and exits.

**Final Comments**

Your source file `net_sim.c` contains a template for the basic functions needed to implement the simulator. The source file also has useful documentation. The function prototypes for the simulator functions are defined in the header file `dlsim.h`.

Happy debugging!