CS 571 Computer Networks **Fall 2006**

Programming Assignment 3: Token Ring Simulation

Version 1.2 *Due 11 December 2006*

1 Overview

In this assignment you will work in groups of two to implement an asynchronous Token-Passing Ring network. Each group will implement a "station" program that communicates via TCP with two other stations. The program receives data on one TCP connection and transmits it on the other. The program also obtains input from a "higher layer" and transmits it in the form of a frame on the ring according to the token-passing protocol specified below. Frames received on the ring that are addressed to the station are delivered to that "higher layer".

The purpose of this assignment is to give you experience with use of asynchronous I/O (i.e. signals or Java threads), and to help you understand the dynamics of token-passing networks and of TCP.

You will work in groups of two to implement this protocol. You may work in C, C++ or Java. Your system *must* run on the Linux machines in the multilab. The goal is to have a group "bakeoff", in which all the implementations are connected together in a ring and cooperate to pass data.

2 Token-Passing Protocol

This section describes the "Medium Access Control" protocol. The goal of the protocol is to allow all stations access to the "medium" in a fair manner, by ensuring that there is at most one frame being transmitted at any time. The latter property is ensured by means of a "token"—a unique frame. Before transmitting a data frame, a station must remove the token from the ring; after transmitting the frame the station must replace the token on the ring. The token format is specified in such a way that flipping one bit in the token converts it into the beginning of a data frame, and vice versa. In the steady state there is at most one token/header byte sequence on the ring at any time. When no data is being transmitted, the ring contains only the token and enough null bytes (i.e. 0x00) to fill the rest of the ring.

The next subsection describes the frame format. Later we describe the byte-level behavior of the station, followed by some parameters of the protocol and their effect on the dynamic behavior of the ring.

2.1 Frame Format

The frame format is shown in Figure 1.

The token consists of the three-byte sequence **start delimiter (SD)**, **control (CTL)**, **end delimiter (ED)**. A data frame consists of SD followed by the control byte (with the appropriate bit set, to distinguish the frame header from the token), followed by the 2-byte destination "MAC address" (most significant byte first), followed by the 2-byte source "MAC address" (most signficant byte first), followed by 0 or more bytes of payload, followed by the end delimiter, followed finally by an **ack** byte. The ack byte provides a low-overhead way for the recipient of a frame to send an acknowledgement back to the originator. The ack byte is transmitted as all zeros by the originator of a frame; when the last byte of the payload is copied at the recipient, it sends a byte of eight ones for the ack byte. Thus, the sender can tell whether the address in the destination field was recognized by any station on the ring.

The protocol requires very little bit-level manipulation. The control byte values are as follows:

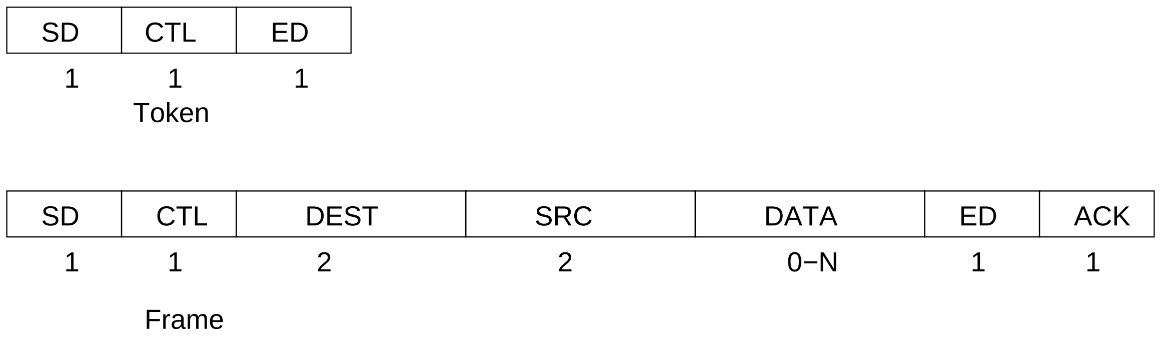

Figure 1: Frame formats; numbers indicate field size in bytes

start delim (SD)	0x11
end delim (ED)	0x14
control (CTL)	0x84 (token)
	0x94 (header)
ack	0x00 (initially)
	0xFF (after frame received)
escape (DLE)	0x10
substitution (SUB)	0x1A

Bit 4 of the control byte[1] is set to zero in a token and set to one in a frame header. Thus, to "grab the token" and start transmitting, a station merely sets bit 4 of the CTL byte of the token as it goes by.

2.2 Payload Stuffing

Because the protocol relies on a unique marker (i.e. **ED**) to mark the end of the frame, the payload must be byte-stuffed; the stuffing technique ensures that ED does not occur in any payload. More precisely, before transmission every occurrence of ED in a payload is replaced by the two-byte sequence **DLE SUB**. To prevent confusion, occurrences of DLE in the payload must be replaced by DLE DLE at the sender. The receiver maps every incoming DLE DLE pair to a single DLE (and ignores that byte's escape function), and every DLE SUB pair to ED.

The stuffing/bytemapping process is transparent to users; the processing is done at the originating station before transmission and removed by the receiving station before delivery. Note that stuffed bytes are *not* removed by stations (including the recipient) who are copying the frame from input to output. Note also that the receiving station does not start searching for ED until it has received two two-byte MAC addresses; thus **the addresses need not be stuffed**.

2.3 Station Behavior: Byte Level

Each station repeatedly reads a byte from its input channel, processes it, and then transmits a byte on its output channel. Thus, the number of bytes on the ring is constant. Stations may read and transmit more than one byte at a time, but logically they must behave as if they handle one byte at a time. Notwithstanding the previous sentence, a station MUST NOT receive more than 16 bytes without transmitting, or transmit more than 16 bytes at a time without receiving. **This rule is very important; without it, the ring will "collapse"**.

In the steady state, with nothing to transmit, the station examines each incoming byte to see if it is the beginning of a header. When the two-byte sequence SD-CTL is recognized, the station checks the next 16 bits (destination address)

[1] Bits are numbered according to the power of 2 they represent, i.e. bit 0 is the low-order bit in a byte and bit 7 is the high-order.

to see whether they match the station's own address. If so, the station records the next 16 bits (source address) and then processes and records each byte of the payload until it sees the end delimiter ED, followed by the ACK byte. The addressed (destination) station sets the appropriate bits in the ACK byte to acknowledge that the frame was received. The station then passes the received payload, along with source address information, to the higher-level protocol. (Note that un-stuffing can be done either as the bytes come off the wire or later, before data is returned to the sender.)

If the destination address in the header of the incoming frame does not match the station's address, it simply transmits each received byte on its outgoing channel without copying it. Once the ending delimiter has been copied, the station goes back to searching for the start delimiter.

When a station has a frame to transmit, it searches for the start delimiter followed by the control byte with bit 4 set to zero. (**Note well** that it must also simultaneously carry out the other parts of the algorithm above, i.e. continues looking for frames addressed to itself). When the station recognizes the SD-CTL sequence, it sets bit 4 of the control byte before transmitting it. It then transmits, in order, the destination MAC address, its own MAC address as the source, the (stuffed) payload, and finally the end delimiter. After transmitting ED, the sender transmits at least one null (all-zero) byte to serve as the ACK byte; then it transmits nulls until it receives the SD-CTL sequence on its incoming channel. If the SD-CTL sequence is received before ED is transmitted, the SD-CTL sequence (with bit 4 of CTL clear) can be re-transmitted immediately after the ACK byte. As long as the sender continues to receive bytes of the frame it transmitted, it re-transmits them as nulls.

2.4 Protocol Parameters

Normally token rings operate synchronously and define certain parameters in units of time. Because the transmission rate is fixed, these parameters are effectively defined in terms of bits (seconds $\times$ bits/second = bits). Since our implementation does *not* have a fixed transmission rate, but operates asynchronously, we define these quantities directly in terms of bytes.

Station Delay. No station may receive more than 16 bytes without transmitting all pending bytes. That is, the maximum "station delay" is 16 bytes. Moreover, a station must transmit **exactly one byte for every byte it reads from its input channel**. (The one exception to this rule is the initial insertion of the token by the ring monitor. In that case the monitor may transmit only until it receives the start delimiter, and thereafter must follow the same rule.) This rule ensures that, once the ring is initialized, the number of bytes in transit on the ring remains constant.

Token Holding Time/Frame Size. Token ring protocols generally specify a maximum Token Holding Time (THT), which bounds the time a station may keep the token (and thus the number of bits it may transmit before having to give up the token). This protocol instead defines a a maximum (unstuffed) frame size, which, given the station delays, should achieve the same effect. The maximum (unstuffed) payload size for this protocol is **4096 bytes**. (Note that the size of the payload "on the wire" (i.e., stuffed) could be as much as 8192 bytes.) Frames may be empty, i.e. the **minimum frame size** is zero.

Number of frames/token rotation. Upon receiving the token, a station may transmit **at most one frame** before passing the token to the next station.

MAC Address. Each station's "MAC address" is the low-order 16 bits of its IP address.

3 Required Structure

Your station implementation *must* have the general structure depicted in Figure 2: four modules, each responsible for a different function:

- Input Section: responsible for interacting with the user, getting input, and passing it to the Transmit Section. The user interface may be implemented in any fashion. One possibility is to have the user input a filename, of the form X-k, where X is a MAC address (i.e. last two bytes of an IP address, which presumably starts with 128.163) and k is a selector to allow multiple frames to be sent to the same address. **The Input Section should not know anything about the token-passing protocol.**

- Transmit Section: responsible for executing the parts of the MAC protocol dealing with data transmission. Transmits payloads passed to it by the Input Section. When a frame needs to be transmitted, looks for the token and when it is received, converts it to a frame header. Byte-stuffs data as it is placed on the "wire". **The Transmit Section should not know anything about the User Interface.**
- Receive Section: responsible for executing the parts of the MAC protocol dealing with data reception. Reads bytes from the incoming TCP connection and passes them to the Transmit Section. Also looks for the station's address and when it is recognized, begins copying incoming bytes to an output buffer, un-stuffing them in the process. **The Receive Section should not know anything about the User Interface.**
- Output Section: responsible for making received payloads available to the user in some way. One possibility is to place received payloads in files with names of the form Y-k, where Y is the source MAC address and k is a per-source counter. **The Output Section should not know anything about the MAC protocol.**

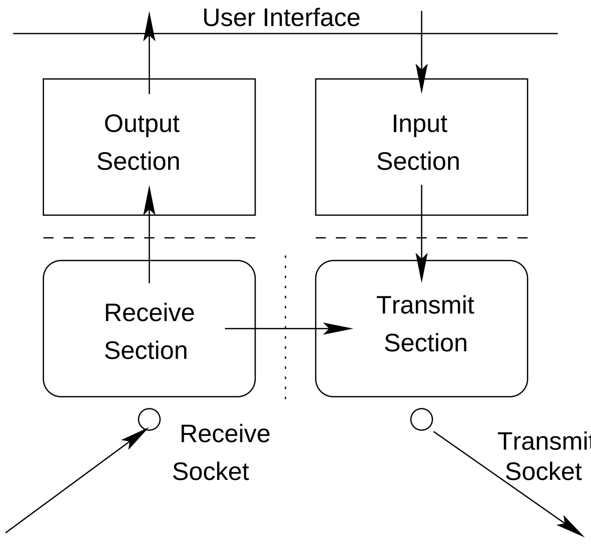

Figure 2: Required Program Structure

Your implementation must also satisfy some requirements that depend on the language you use to implement it. In particular, the way the different sections interact with each other is determined by the language you will use. (**Note:** The entire project must be done using the same language.)

3.1 C/C++ Requirements

If you use C/C++, your implementation *must* be structured as follows. The Input Section, Output Section, and Transmit Section are separate Processes. The Receive Section is implemented as a single function that is the signal handler for SIGIO. That is, when data arrives on the incoming TCP connection, SIGIO is invoked, and the Receive Section runs.

The Input Section communicates with the Transmit section via a *pipe*. Payloads are written to the pipe by the Input Section, and read from it by the Transmit Section. Similarly, the Output Section communicates with the Receiver Section via a pipe. Because a pipe is a byte-stream channel, it is necessary to define a *framing protocol* for use on the channel. One possibility is to prepend to each payload a header containing two bytes of length and two bytes of MAC address (source or destination, depending on which section is creating the frame).

As noted above, the Receive Section runs as an interrupt-level thread in the same process as the Transmit Section. Because it is at Interrupt level, it must never block. Therefore it must use *non-blocking I/O*. The Receive Section passes bytes to the Transmit section via a buffer.

The general order of steps of your program should be:

1. Read configuration file to get parameters (e.g. address of downstream neighbor to connect to).
2. Create pipes for user interface.
3. Fork(); close unused file descriptors in children; execute UI in children.
4. In the parent (Transmit Section), create listening socket, bind it to port 57103, and arrange for SIGIO to be delivered upon a connection. Establish SIGIO handler using the `sigaction()` system call (or its equivalent).
5. Transmit Section: create a socket and open a connection to downstream neighbor.
6. Set up interface between Transmit and Receive Sections.
7. Receive Section: when a connection comes in from upstream, accept it and signal the Transmit Section that it is ready to go.
8. If this station is the Ring Monitor, place the token on the ring along with an appropriate number of fill bytes. (Transmit Section)

3.2 Java Requirements

If you use Java, your program *must* be structured using a separate thread for each section. The interface between the User Interface and MAC sections can be anything, but each thread *must* actually do something, i.e. it cannot simply be a holder for methods called by other threads that do all the work. The Transmit and Receive threads must communicate using a buffer with capacity of at most 16 bytes. (You do not have to use the Java class `Buffer`, but you may if you want to.) Threads must use appropriate mutual exclusion mechanisms to ensure safety of any shared data structures.

The Java program must execute the following steps:

1. Open and read the configuration file to get parameters (e.g. downstream neighbor address).
2. Create and start the appropriate thread for each section.
3. Receiver Section: create a listening socket for incoming connections and bind it to port 57103. When connection arrives, accept it and inform Transmit Section it is ready to go.
4. Transmit Section: create a socket connected to downstream neighbor (port 57103).
5. Transmit Section: if this station is the monitor, place the token on the ring, followed by an appropriate number of fill bytes.

4 Additional Capabilities

In addition to performing the basic functions specified above, your implementation **should** also have the following capabilities:

- Keep track of the time between visits of the token to the station, and print the average and latest measured values each time the token is “seen” at the station. (Note that a frame header should be considered a token for the purposes of this measurement.)
- Measure the throughput attainable on the ring, by measuring the time it takes to transmit a large frame all the way around the ring, from the time the first byte is placed on the ring until the last byte is removed.

You will demonstrate your program in a classwide “bake-off,” to be held sometime during finals week.