CS 536 (Park) **Assignment III** **Due: Nov. 6 (Tue.), 2001**

PROBLEM 1 (30 pts)

Consider the 2-D tiling problem using hexagons of various colors. Give a proof that 7 colors suffice. What about 3 colors? Either way, provide a proof that 3 colors suffice, or that they do not.

PROBLEM 2 (30 pts)

Assume our goal is to turn a CSMA/CD Ethernet into a token ring. That is, while preserving Ethernet's existing MAC, we want to build a system—in software—that emulates a token ring-based access mechanism with its associated deterministic properties. How would you go about doing that? Give a detailed sketch of a design and explain how it is able to function as a token ring.

PROBLEM 3 (70 pts)

As a continuation of Problem 2, implement a token ring architecture with the following reduced functionality. First, the token ring daemon, `tokenringd`, is executed on participating hosts with

% `tokenringd` *wait-time IPaddr-1 IPaddr-2 ... IPaddr-n*

where *IPaddr-1* $\rightarrow$ *IPaddr-2* $\rightarrow \cdots \rightarrow$ *IPaddr-n* $\rightarrow$ *IPaddr-1* forms a ring of n hosts. Upon start-up, each process waits for a uniformly random time between 0–*wait-time* msec before instituting leader election. A process institutes leader election by sending a special control message—"ARE-YOU-ALIVE"—to its next neighbor node (specified by the command-line ring structure) which, if alive, will respond with an acknowledgment message "WHAT-DO-YOU-THINK". If no acknowledgment is received within *wait-time* msec, the same query is repeated with the next node down the line, until an alive node is identified—designated the effective neighbor—or all of the nodes are deemed to be down, at which point the daemon dies out of loneliness.

Upon identifying the effective neighbor—if all nodes are up, the effective neighbor of node i should be $i + 1$ mod $n + 1$—a message is sent to the effective neighbor with its 4-byte IP address inscribed in the payload. The first two bytes of the payload are used to encode the type of the message: 00 "IDLE TOKEN", 10 "ARE-YOU-ALIVE", 01 "WHAT-DO-YOU-THINK", 11 "LEADER ELECTION", 22 "CONFIRM LEADER". Upon receiving a "LEADER ELECTION" message from a neighbor, the token ring daemon appends its own IP address at the end of the payload and passes the message toward its effective neighbor. When a node receives a "LEADER ELECTION" control packet which was initiated by itself—the first IP address inscribed in the payload is its own—then it instantiates a "CONFIRM LEADER" message propagation by forwarding the message again down the chain, albeit with the type field having value 22. The node with the lowest IP address value—upon receiving a "CONFIRM LEADER" message—is deemed the leader. It stops forwarding the "CONFIRM LEADER" message, and introduces a 00 type token into the ring with empty payload. The type 00 token will circle around and around until such time when a node crashes or the token packet is dropped by the network. To prevent more than one type 00 token from being introduced into the network, an elected leader will ignore all "CONFIRM LEADER" messages following the first "CONFIRM LEADER" is has received for *wait-time* msec. Each node starts a timer after processing a token message, whatever its type, which goes off after *wait-time* msec. If a token is received within *wait-time* msec, the timer is reset again. If a token is not received with *wait-time* msec, a new leader election is initiated by following the same procedure followed at start-up. All packets should be UDP packets, and choose a port number to be used by `tokenringd`.

Perform benchmarking with 3 nodes, 5 nodes, 7 nodes, 9 nodes, 11 nodes, with *wait-time* msec set a 2000 msec. A daemon, upon performing a token operation, whatever its type, should print the type of operation, the token contents, and the time of day. If the token is of type 00, it should print out, in addition to the time of day, the measured token rotation time which is the time interval from the last received type 00 token until the current one. A daemon, from start-up, counts how many tokens it has handled, and terminates when the count reaches 30. Submit hardcopy output of the benchmark runs, and argue why the output trace shows the correct functioning of the system.

Perform the 11 node benchmark with one of the nodes running a "faulty" daemon, which upon receiving a token, flips a coin, and with probability p "drops" the token (forgets to forward it). Print a message of the drop event when it occurs. Run the benchmark with $p = 0.1$, 0.3, 0.5, and 0.7 (only one daemon is faulty). What do you observe?

Perform the 11 node benchmark with all faulty daemons, but instead of dropping a token with probability p, let the daemon terminate with probability p instead. Print a message indicating imminent termination before performing `exit`. Run the experiments with $p = 0.01$, 0.05, and 0.1. From the output traces, explain how your system is recovering.

PROBLEM 4 (70 pts)

Implement an end-to-end sender/receiver application pair using UDP where the sender process, `udp-send`, on the source host transmits a sequence of UDP packets to the receiver process `udp-receive` on the destination host. `udp-send` takes two command-line arguments

% `udp-send` *interarrival-time session-duration*

where *interarrival-time* is the time interval between successive packets (measured in milliseconds) and *session-duration* is the total session duration or lifetime (i.e., after *session-duration* seconds from start-up `udp-send` is to terminate), measured in seconds. The receiver, `udp-receive`, takes a single command-line argument, *interservice-time*, which is the time duration (or "idle time") between successive servicing of two packets.

The receiver works by responding to two signals—SIGPOLL and SIGALRM—in an asynchronous manner using signal handlers. When a SIGPOLL interrupt occurs, the receiver's SIGPOLL handler fetches the newly arrived UDP packet(s) from the kernel's buffer space and enqueues the packet into its own user space buffer, implemented as a first-come-first-served (FCFS) queue. For tracing purposes, the SIGPOLL handler also takes a time stamp using `gettimeofday`() and records both the time stamp and the current queue length—excluding the newly arrived packet—into a LOG file called *log-queue* for later examination. If the user space buffer, at the time of packet arrival, is full, then the UDP packet is dropped or discarded. This event is then also recorded using a time stamp into a separate LOG file for packet drops called *log-drops*. When a SIGALRM interrupt occurs (signalling that a packet should be serviced), the receiver's SIGALRM handler dequeues a UDP packet from its FCFS queue. Before exiting the handler, the interservice-time parameter is used to set when to wake up next.

Test your application using the following set-up. Use a fixed UDP packet payload of size 1kB (containing an all zero bit pattern) and a user space buffer capacity of 300 MTUs (i.e., 300kB). For interservice-time 200ms, run six experiments with interarrival-times 350ms, 250ms, 200ms, 150ms, 100ms, 50ms for a fixed session duration of 30s. For each run, plot two time series (or graphs) using the log files *log-queue* and *log-drops*, one showing the queue length as a function of time and the other showing packet drops as a function of time. Give your interpretation of the queueing behavior and packet drop behavior as the interarrival time is increased.

Note: An important programming consideration here is the management of the two interrupts such that they do not get in "each other's way." For example, while the SIGALRM signal handler is active, the SIGPOLL signal should not be able to interrupt the SIGALRM signal handler—and vice versa—as this can lead to a corrupted state of the user space buffer which is manipulated by both. As part of the hand-in, explain in half a page of write-up how your program manages the orderly access to the user space FIFO queue by the two signal handlers.