CS 422 (Park) **Assignment V** **Due: Apr. 30 (Fri.), 2004**

<u>Code submission instructions:</u> *For code implementations, organize your code around a Makefile that automates compilation. All source and object code should be put in subdirectories. Use subdirectories to organize your output files. The current directory should be empty except for Makefile, a README file, and subdirectories. The README file should briefly describe what is contained in the subdirectories. Makefile should create the executable(s) in the current directory. Your code will be tested by automated scripts that check compilation and run-time behavior. Your source code will be tested manually and by automated scripts. Consult the TA's link for any additional instructions.*

PROBLEM 1

Read Chapters 27, 31, 33, 39, 40, and 41 from Comer.

PROBLEM 2 ($55 = 40 + 15$ pts)

(a) Use the simple "talk" (i.e., chatting) application in `pub/cs422/cs410/Lec17+` comprised of client/server codes with hardcoded IP addresses and port numbers as a template to create a single peer-to-peer chat application, call it `my_chat.c`. In a peer-to-peer application, there is no distinction between client and server. The executable, `my_chat.bin`, takes as command-line input a sequence of IP address (in symbolic form) and port number pairs

IP-addr-1 port-1 IP-addr-2 port-2 $\cdots$ *IP-addr-n port-n*

where the addresses indicate the parties with whom one wishes to talk (a "chat room" or n-party conference call). Implement a simple form of multicast where a message that one types in is sent, one-by-one, to each destination address (done by a loop). A received text message, when shown on the terminal, should be preceded by the sender's IP address and port number. Use your programming skills and creativity to devise an output (and input) format that keeps the screen uncluttered when messages are typed in and received from multiple parties. Grab a couple of friends and test your application on three separate machines. Use `script` to record the session.

(b) Write a 1–2 page technical explanation of how the chat application works, with focus on the control flow of the program with respect to the role played by asynchronous I/O, the blocking nature of `fgets`(), and interrupts. Discuss why asynchronous I/O is required, how multiple interrupts generated by multiple arriving messages are handled to prevent one message printout from being garbled by the arrival of another. What about garbling of a message that the local user types in? Discuss how you would restructure your code to implement it using TCP (UDP, after all, is unreliable) and the pros/cons vis-à-vis the UDP based implementation.

PROBLEM 3 (70 pts)

As a continuation of Problem 3, Assignment IV, modify the CBR sender and receiver code so that they perform adaptive congestion control. The receiver maintains a buffer—a FIFO queue—into which arriving packets (i.e., their payload) are stored. The receiver dequeues from the buffer an item every γ microseconds trigged by SIGALRM, raised by `ualarm`() and caught by a signal handler that performs the dequeue operation. It's important that the queue not get corrupted by SIGALRM and SIGPOLL signal handlers operating on the data structure at the same time. Use signal blocking to achieve this goal without resorting to semaphores or locks. In addition to the port number on which it waits, `cbr_receiver` takes two additional command-line arguments, the buffer dequeue time interval γ (μsec) and a target buffer occupancy Q^* (byte). The SIGALRM handler, upon dequeueing an item, takes a time stamp and logs the time stamp and current buffer occupancy (i.e., queue length) $Q(t)$ in a separate log file. The SIGALRM handler also sends a feedback ("ACK") packet back to the sender where the payload of the UDP packet is 4 bytes containing the integer value: $Q(t) - Q^*$. The SIGPOLL handler remains the same as before except that it enqueues newly arriving packets into the buffer.

The sender, `cbr_sender`, stays the same except that it adjusts the initial *packet-spacing* based on the feedback received from `cbr_receiver`. It has a SIGPOLL handler that updates *packet-spacing*, call it τ, by the following rule: if the feedback value (i.e., $Q(t) - Q^*$) is negative (buffer occupancy is low), decrease τ by ε (which speeds up the transmission) where $\varepsilon > 0$ (μsec granularity) is an additional command-line argument. Ensure that τ stays nonnegative. Conversely, if the feedback value is positive, increase τ by ε. The SIGPOLL handler logs the time stamp and updated τ value in a separate log file. The logs should reside in memory and be flushed to disk at the end of the run to reduce slow-down due to disk I/O.

Benchmark your application using two `xinu` machines for sender and receiver. The receiver's additional parameters are $\gamma = 100000$ μsec and $Q^* = 30$ KB (the receiver's total buffer size should be sufficiently large, say 100 KB, hardcoded in the application). The sender's parameters are: *packet-size* 1 KB, *packet-count* 600, (initial) *packet-spacing* $\tau = 80000$ μsec, and adjustment parameter $\varepsilon = 10000$ μsec. In addition to the sender and receiver data rate plots, use `gnuplot` to plot the buffer occupancy evolution at the receiver and the *packet-spacing* changes at the sender. Repeat the benchmarks with $\varepsilon = 15000$ μsec and 5000 μsec. Does congestion control perform better, worse, or the same? Give a 1/2-page discussion of your results.

PROBLEM 4 (50 pts)

A variation of Problem 3, Assignment IV, that implements source routing, modify `cbr_receiver` into a router, call it `my_router`, that takes a port number as command-line argument as before. Upon receiving a packet in the SIGPOLL signal handler, it inspects bytes 4–8 of the payload (bytes 1–4 contain a sequence number) which contains a positive integer, call it i. It then jumps to the $(9 + 6(i - 1))$'th byte in the payload. The payload following the first 8 bytes is organized into $6 = 4 + 2$ byte blocks where the first 4 bytes contain the IP address of the next hop and the following 2 bytes the next hop's port number. The router's SIGPOLL handler crafts a new UDP packet with the IP address and port number indexed by i, increments i in the payload (so that the next hop router can correctly look up the $(i+1)$'th hop), and sends the packet on its way. It also prints onto the screen the current time, host IP address and port number, packet's sequence number, destination IP address and port number, source IP address and port number, and i. The sender, `cbr_sender`, is slightly augmented so that in addition to the other command-line arguments, it takes a sequence of IP address and port number pairs (in a format similar to Problem 2) that specify the route that a packet tranmitted by the sender should take. On the payload of each packet, the first 4 bytes contain the sequence number as before, bytes 5–8 the value 1, and the following 6-byte blocks the IP address and port number pairs starting from the second pair. The first pair contains the "default" router's IP address and port number that the sender uses to address the packet. The last pair contains the final destination, i.e., the host where `cbr_receiver` runs. The code of `cbr_receiver` does not change at all.

Benchmark your application using 7 `xinu` machines for sender, receiver, and 5 routers. Indicate a specific sequence of hops that packets should take in the command-line argument of the modified `cbr_sender`. As for the other parameters, set them to *packet-size* 1 KB, *packet-count* 20, and *packet-spacing* 500000 μsec. Use `script` to record the output at the routers and sender. Plot the sender and receiver packet logs.

EXTRA CREDIT PROBLEM (30 pts)

A continuation of Problem 3, implement congestion control methods B, C, and D discussed in class, and compare their performance. It is up to you to choose the new congestion control parameters and make them part of an expanded command-line argument set. Show the buffer occupancy and *packet-spacing* plots. The aim is to demonstrate that Method D indeed works best. Provide a 1-page discussion of your results.