CS 536 (Park) **Assignment I** **Due: Oct. 2 (Tue.), 2001**

PROBLEM 1 (30 pts)

Based on our discussion and understanding to date regarding what a computer network is, explain the similarities and differences between a computer network and an airline network (airplanes, passengers, airports, service providers). Focus on essential functionalities and features which may ultimately affect performance. This implies inclusion of performance metrics in the discussion. Your discussion should include explicit correspondences such as "packet $\leftrightarrow$ passenger" or "packet $\leftrightarrow$ airplane," and your criteria in making choices when the correspondences are not necessarily unique. Discuss what factors delay packets in computer networks, and what factors do so in airline networks. In your opinion, which type of network seems easier to manage? Give your reasoning.

PROBLEM 2 (100 pts)

Consider a Direct Sequence CDMA system where we are interested in transmitting binary messages belonging to multiple users. Assume that the code rate of the Direct Sequence CDMA (DS-CDMA) system is 4 (i.e., each bit or symbol is replicated 4 times; note that the examples discussed in class used a code rate of 5). For example, if an user wanted to send the message 101, we would first expand it to

111100001111

before doing any further processing/encoding.

We will assume that our DS-CDMA system uses 1 and -1 in place of 1 and 0, that is, "1" is mapped to "1" and "0" is mapped to "-1". Since the code rate is 4, when using orthogonal chip sequences—the $\mathbf{y}^i = (y_1^i, y_2^i, y_3^i, y_4^i)$ vectors belonging to different users i discussed in class—we know that we can support up to 4 concurrent users while guaranteeing correct decoding. Let the chip sequence set $\mathcal{Y} = \{\mathbf{y}^1, \mathbf{y}^2, \mathbf{y}^3, \mathbf{y}^4\}$ be given by

$$\mathcal{Y} = \{\, (1,1,1,1), (-1,-1,1,1), (-1,1,-1,1), (-1,1,1,-1)\,\}.$$

You should make sure to verify that the chip sequence set is pairwise orthogonal, i.e., for any two distinct chip sequences $\mathbf{y}^i = (y_1^i, y_2^i, y_3^i, y_4^i)$ and $\mathbf{y}^j = (y_1^j, y_2^j, y_3^j, y_4^j)$, $i \neq j$, we have

$$\mathbf{y}^i \circ \mathbf{y}^j = y_1^i y_1^j + y_2^i y_2^j + y_3^i y_3^j + +y_4^i y_4^j = 0$$

where "$\circ$" denotes the dot product.

Encoding Assume that each user wishes to transmit a single bit—if she wishes to transmit a thousand bits, we would repeat the procedure to be described presently one thousand times—and let user i's ($i = 1,2,3,4$) 4-fold replicated as well as $\{1,-1\}$-mapped data sequence be denoted by $\mathbf{x}^i = (x_i, x_i, x_i, x_i)$ where $x_i \in \{1,-1\}$. For example, if user i wished to transmit "0" then $\mathbf{x}^i$ would be $(-1,-1,-1,-1)$. We will define a new bit-wise multiplication of two vectors, denoted by "$\star$", as

$$(a,b,c,d) \star (e,f,g,h) = (ae, bf, cg, dh)$$

where "$a\,c$" is our "plain old" multiplication of two real numbers a and c. The code vector of user i, denoted $\mathbf{z}^i$, is computed as

$$\mathbf{z}^i = \mathbf{x}^i \star \mathbf{y}^i = (x_i, x_i, x_i, x_i) \star (y_1^i, y_2^i, y_3^i, y_4^i) = x_i (y_1^i, y_2^i, y_3^i, y_4^i) = x_i\, \mathbf{y}^i.$$

Lastly, having computed the $\mathbf{z}^i$'s for each user i, we arrive at the final code vector $\mathbf{Z}$ that will be transmitted

$$\mathbf{Z} = \mathbf{z}^1 + \mathbf{z}^2 + \mathbf{z}^3 + \mathbf{z}^4 = x_1\, \mathbf{y}^1 + x_2\, \mathbf{y}^2 + x_3\, \mathbf{y}^3 + x_4\, \mathbf{y}^4.$$

Important: the summation above is our "plain old" summation component-wise summing of two real numbers (i.e., we are not using modulo arithmetic). Thus, $\mathbf{Z}$ is a vector whose components can be $0, \pm 1, \pm 2, \ldots$ (a slight correction from what we discussed in class). Now, to decoding.

Decoding Each receiver will receive the same "jumbled" code vector $\mathbf{Z}$, and has to recover what his sender sent. The receiver corresponding to user i will perform the operation $\mathbf{Z} \circ \mathbf{y}^i$. Thus, the receiver is assumed to know the chip sequence $\mathbf{y}^i$. Since the above is a dot product of two vectors, it yields a scalar, call it r_i. We can determine what the value is as follows:

$$r_i = \mathbf{Z} \circ \mathbf{y}^i = (x_1\, \mathbf{y}^1 + x_2\, \mathbf{y}^2 + x_3\, \mathbf{y}^3 + x_4\, \mathbf{y}^4) \circ \mathbf{y}^i = x_i (\mathbf{y}^i \circ \mathbf{y}^i) = 4 x_i.$$

The simplification of $\mathbf{Z} \circ \mathbf{y}^i$ to $x_i(\mathbf{y}^i \circ \mathbf{y}^i)$ follows from orthogonality of the chip sequences, and the last step follows from our specific choice of the chip sequence (you should verify that $\mathbf{y}^i \circ \mathbf{y}^i = 4$ for all i). Thus, if $x_i = 1$ then the receiver receives $r_i = 4$, a 4-fold amplification of the original value, and analogously in the opposite direction yielding $r_i = -4$ for $x_i = -1$.

So, what's the homework? First, make sure to fully understand how, and why, our specific version of DS-CMDA works. Second, implement the encoding and decoding functions in C. The function definition for the decoder—for a single data bit—is given by

int `ds_cdma_decode_bit`(int *id*, int *rate*, int $*z$)

where *id* is the user ID (in our case, 1–4), *rate* is the code rate (in our case, 4), and *z*[ ] is an array containing the received "signal" $\mathbf{Z}$ (in our case an array of length 4). `ds_cdma_decode_bit( )` returns the decoded value r_i corresponding to *id*; thus r_i hasn't been "cleaned up" and mapped back to 0 or 1. The function definition for the encoder—again for a single data bit—is given by

int `ds_cdma_encode_bit`(int *id*, int *rate*, int x, int $*z$)

where *id* is the user ID, *rate* the code rate, x the data bit (0 or 1), and *z*[ ] an array of length *rate* where the $\mathbf{z^i}$ corresponding to user *id* is to be stored. The specific chip sequences are to be defined in a header file *ds_cdma.h*. We also define a utility function

int `ds_cdma_multiplex_bit`(int *rate*, int $*z1$, int $*z2$)

where *rate* is the code rate, and *z1*[ ], *z2*[ ] are arrays of length *rate*. `ds_cdma_multiplex_bit( )` simply adds *z2*[ ] onto *z1*[ ] (overwriting latter's value). Thus, this function can be used to calculate the final code vector $\mathbf{Z}$ (among other things). Put each function in a separate file of the same name.

Third, to put the system to the test, write driver code that will test the correct functioning of the CDMA encoding/decoding for the 16 possible concurrent single data bit transmissions by 4 users. For example, 0000 means that all 4 users are sending 0's, 0110 would mean that users 1 and 4 are sending 0's while users 2 and 3 are sending 1's. For each such combination, output/print $\mathbf{z^1}, \mathbf{z^2}, \mathbf{z^3}, \mathbf{z^4}$, $\mathbf{Z}$, and r_1, r_2, r_3, r_4.

Fourth, consider the case where $\mathbf{Z}$ is being transmitted over a wireless medium (say, open space), and it happens to be raining which introduces noise into the transmitted "signal" $\mathbf{Z}$ (the electromagnetic waves representing $\mathbf{Z}$ will be "colliding" with falling rain droplets, which can't be good). Suppose that the end result of this is that the actual signal received, call it $\hat{\mathbf{Z}}$, is a deformed version of $\mathbf{Z}$, and has the form

$$\hat{\mathbf{Z}} = \mathbf{Z} + \mathbf{n}$$

where $\mathbf{n}$ is a "noise vector" whose components are $0, \pm 1, \pm 2, \ldots$. For example, $\mathbf{n} = (0, 0, 0, 0)$ would mean that there was no noise, and $\mathbf{n} = (1, 0, 0, -2)$ would mean that the magnitude of the first component was increased by 1 and the magnitude of the last component was decreased by 2 (rain works in mysterious ways). To test the resilience of our DS-CDMA scheme against noise, use `ds_cdma_multiplex_bit( )` with the second argument as $\mathbf{n}$ to measure the impact of all possible noise vectors when each component is restricted to be $0, \pm 1, \pm 2$ (there are 5^4 possible combinations). For each noise vector, count how many of the 4 users end up being impacted by incorrect decoding (e.g., in the case of data bit 0, receiver i computing $r_i \geq 0$ leads to incorrectness). Depict DS-CDMA performance with respect to noise resilience using a 2-D plot, where the x-axis ranges over the noise vector combinations in canonical order—i.e., $(-2, -2, -2, -2)$, $(-2, -2, -2, -1)$, $(-2, -2, -2, 0), \ldots, (2, 2, 2, 2)$—and the y-axis shows the number of users with unsuccessful decodings (0–4). Use the utility `gnuplot` with the style option *impulse* to plot the output (you may use other plotting software if they are of comparable quality). Perform the above for three possible user inputs 0000, 1001, 0101 (out of the 16 total combinations).

Fifth, what does the plot tell you, and how would you evaluate the effectiveness our DS-CDMA scheme with respect to its noise resilience based on the pattern (if any) seen in the plot? Which noise vectors are worst, and which are the least intrusive? Among the noise vectors that are least (or less) damaging, are there those whose components comprise of just -1 or 1 (identify them). For what other important positive purpose might they be used?

PROBLEM 3 ($80 = 20 + 30 + 30$ pts)

(a) Explain the notions *atomicity*, *blocking/non-blocking*, *synchronous/asynchronous I/O*, *mutual exclusion and semaphores*, and *interrupts*, and in which system calls they play a role.

(b) You will find a client/server application under ~park/pub/cs536; the client program is `client.c` and the server program is `server.c`. Reverse-engineer the code and explain what the server and client are doing. Then, give a line-by-line explanation of the code—you may group several lines into one unit, as you see fit, and give a modular description. The important criterion is that your answer makes clear that you understand all aspects of the code. If you find inefficiencies, indicate what and why, and suggest possible changes. Important: include in your answer a clear explanation of what trick is being used to direct the output of the server back to the client and how this works. Compile the code and get it to run. What do you observe? What happens when you replace `fortune` with `finger`?

(c) Write a program that uses the SIGALRM signal to periodically update a counter. The counter is initialized to zero, and incremented by 1 each time SIGALRM goes off. Your program, call it `wakeup`, should take a command-line argument

% `wakeup` *period*

where *period* is a positive integer that specifies the time period between successive wake-ups/counter updates in milliseconds (units). The main body of the program is simplicity itself: after registering the signal handler, it uses `ualarm( )` to set the first alarm to go off after *period* milliseconds—to get things started—then enters into an infinite loop where it waits for a signal event to occur using `pause( )`. The SIGALRM signal handler, call it `my_sigarlm_handler( )`, performs two tasks when invoked: update of the counter and call to `ualarm( )` to set up the next alarm before returning. To prevent the program from running forever, we will institute an overall time limit as follows: after registering the signal handler, `gettimeofday( )` is called to remember the starting time. Inside `my_sigarlm_handler( )`, between the counter update and call to `ualarm( )`, `gettimeofday( )` is called to establish the time elapsed since the very first call to `gettimeofday( )`. If the total time elapsed exceeds 60 seconds, the process terminates after outputting the counter value.

Perform a benchmark experiment as follows: run `wakeup` for *period* values 1000, 700, 400, 100, 50, 30, 10, 5, 1, with two runs per *period*. Plot the counter values as a function of *period*, where the pair of counter values per period are plotted as errorbars (e.g., using `gnuplot` with plot style *errorbars*). On the same plot, show the graph corresponding to 60,000 / *period*, which will depict an idealized number of counter updates during a 60 second interval. Comparing the shape of the two graphs in terms of their sensitivity to changes in *period*, what do you observe? List as many potentially relevant the factors—principally an OS question—that may cause the observed phenomenon, in decreasing order of their importance.