CSE 220: Systems Fundamentals I

Stony Brook University
Homework Assignment #1
Spring 2024
Due: Sunday, February 18th, 2024 by 9:00 pm EST

Updates to this Document

- None yet!

Learning Outcomes

1. Perform byte-level computations.
2. Use the bit-level operations to perform computations.
3. Traverse and manipulate an array of bytes.

The ArrayNet Protocol

For this assignment you will be working with a fictional, poorly-designed network protocol called ArrayNet. The protocol is intended for transmitting only arrays of 32-bit signed integers between two hosts. An array can be split into several packets by the sender and then recombined at the destination. Packets might arrive out of order at the destination, and could even become corrupted during transmission. An ArrayNet packet consists of a 16-byte **header** with the fields shown below, plus the **payload** (the actual integers of the array). All fields are unsigned integers.

Byte #

0	1	2	3	4	5	6	7
Source Address				Destination Address			SP | DP

Byte #

8	9	10	11	12	13	14	15
Fragment Offset		Packet Length		MH	Checksum		C | TC

Byte #

16	17	18	19	20	etc.		
Payload of signed integer data (at most 2^{14} - 16 bytes)							

The 10 fields in the header are:

1. Source Address: 28-bit network address of the transmitting host
2. Destination Address: 28-bit network address of the receiving host
3. Source Port (“SP” in the diagram above): 4-bit network port of the transmitting host
4. Destination Port (“DP): 4-bit network port of the receiving host
5. Fragment Offset: 14-bit byte offset of the payload’s integer in the original array
6. Packet Length: 14-bit total length of the packet in bytes, including the header and payload
7. Maximum Hop Count (“MH”): 5-bit field providing the maximum number of routers the packet may visit while traveling from the source to the destination.
8. Checksum: the 23-bit remainder of dividing the sum of:
    a. all of the fields (except for the Checksum itself) and
    b. the absolute values of the 32-bit integers in the payload

   by the divisor 2^{23}-1. Again, this is a fictional network protocol. This checksum calculation will accept corrupted payloads!
9. Compression Scheme (“C”): 2-bit field providing the ID # of the compression scheme used to compress the payload.
10. Traffic Class (“TC”): 6-bit field used to indicate the quality of service required by this packet.

For instance, consider a packet with the following fields and payload, given in decimal,

```
Source Address: 1908874
Destination Address: 11801002
Source Port: 12
Destination Port: 13
Fragment Offset: 0
Packet Length: 28
Maximum Hop Count: 21
Checksum: 6204849
Compression Scheme: 3
Traffic Class: 51
Payload: 8675309 -39281 92832146
```

This data would be represented as an ArrayNet packet in hexadecimal as the following bytes:

```
0x01 0xd2 0x08 0xa0 0xb4 0x11 0xaa 0xcd  (Header, first 8 bytes)
0x00 0x00 0x01 0xca 0xde 0xad 0xb1 0xf3  (Header, second 8 bytes)
0x00 0x84 0x5f 0xed   (Payload starts here)
0xff 0xff 0x66 0x8f
0x05 0x88 0x81 0x92
```

To help see why these bytes comprise the packet, below are the packet's fields again, this time with the contents in binary. Each decimal value (in purple) is the byte number. Fields are alternatively colored in red and blue. The payload is shown in black.

```
   0        1        2        3        4        5        6        7
00000001 11010010 00001000 10100000 10110100 00010001 10101010 11001101
00000000 00000000 00000001 11001010 11011110 10101101 10110001 11110011
00000000 10000100 01011111 11101101
11111111 11111111 01100110 10001111
00000101 10001000 10000001 10010010
```

Complete the functions specified below in the file `hw1.c` provided in the template repository. Any changes made to `hw1.h` will be discarded. Only the contents of `hw1.c` will be graded.

Template Repository

Starter code is available on CodeGrade (and GitHub), as with previous assignments.

Functionality to Implement

Part 1: Print the Fields and Payload of an ArrayNet Network Packet

```
void print_packet_sf(unsigned char packet[])
```

Given an ArrayNet network packet in the argument `packet`, the `print_packet` function prints the fields in the order shown as in the example in the previous section, ending each line with a newline. It prints all fields in decimal. The function may not print blank lines, extra spaces or extra newlines, labels, or any other output. You may not use the `ntohl()` function or similar built-in functions to perform any data conversions for you. For the payload, combine four-byte chunks to create 32-bit signed integers. Note carefully how the bytes of each integer are ordered in the payload.

For a sample of what the output should look like, see the example in the section named "The ArrayNet Protocol", above.

Part 2: Compute a Packet's Checksum

```
unsigned int compute_checksum_sf(unsigned char packet[])
```

Given an ArrayNet network packet in the argument packet that is valid and complete except for its Checksum field, compute and return the packet's checksum. Do not modify the packet.

Part 3: Reconstruct an Array Stored as a Set of Packets

```
unsigned int reconstruct_array_sf(unsigned char *packets[],
     unsigned int packets_len, int *array, unsigned int array_len)
```

Complete the `reconstruct_sf` function, which takes an array of ArrayNet network packets, extracts the payloads, and writes the original transmitted data into the `array` argument. The function ignores (i.e., skips over) any packets with invalid checksums (i.e., the checksum field in the packet does not match the checksum value computed by the `checksum_sf` function). We call these "corrupted" packets. You may not assume that `array` is large enough to store the reconstructed data. Rather, store up to `array_len` integers. Do not modify any of the bytes of `array` except to write the payloads. The Fragment Offset field in each packet tells you where in `array` to start writing the payload. For instance, if the Fragment Offset of a packet is 40, then you should start writing its payload at index 10 of `array`. (Remember that the Fragment Offset is in units of bytes, not `int`s.) In a real network protocol, packets may be received out of order. It is necessary to always read the Fragment Offset. The function returns the number of integers it was able to write into `array` after discarding any corrupted packets.

Arguments:

- `packets`: an array of character arrays representing packets, i.e., `packets[0]` is a pointer to the first character array, `packets[1]` is a pointer to the second character array, etc.
- `packets_len`: the number of character arrays pointed to by `packets`
- `array`: an uninitialized region of memory where the reconstructed integers must be written
- `array_len`: the length of `array`. This length is given in the number of 32-bit integers that can be stored in `array`, not the size of `array` in bytes.

Example

Suppose the original 18-element array that was packetized was:`{10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150, 160, 170, 180}`. It was

packetized into 5 packets, each carrying a payload of 16 bytes (i.e., 4 integers each), except for the last packet, which has 8 bytes.

The packets arrived out of order (i.e., 4, 0, 3, 1, 2), and packets #1 and #4 were corrupted during transit. The `packets` array contains the following packets, where the packet number given indicates the original order of the packets. Some of the data corruption is highlighted below in red.

Packet #4 Source Address: 12346 Destination Address: 964321 Source Port: 12 Destination Port: 3 Fragment Offset: 64 Packet Length: 24 Maximum Hop Count: 16 Checksum: 977194 Compression Scheme: 0 Traffic Class: 58 Payload: 170 42164	Packet #0 Source Address: 12346 Destination Address: 964321 Source Port: 12 Destination Port: 3 Fragment Offset: 0 Packet Length: 32 Maximum Hop Count: 16 Checksum: 976888 Compression Scheme: 0 Traffic Class: 58 Payload: 10 20 30 40
Packet #3 Source Address: 12346 Destination Address: 964321 Source Port: 12 Destination Port: 3 Fragment Offset: 48 Packet Length: 32 Maximum Hop Count: 16 Checksum: 977416 Compression Scheme: 0 Traffic Class: 58 Payload: 130 140 150 160	Packet #1 Source Address: 16378 Destination Address: 964321 Source Port: 12 Destination Port: 3 Fragment Offset: 16 Packet Length: 32 Maximum Hop Count: 16 Checksum: 977064 Compression Scheme: 0 Traffic Class: 58 Payload: 50 60 70 80
Packet #2 Source Address: 12346 Destination Address: 964321 Source Port: 12 Destination Port: 3 Fragment Offset: 32 Packet Length: 32	

```
Maximum Hop Count: 16
Checksum: 977240
Compression Scheme: 0
Traffic Class: 58
Payload: 90 100 110 120
```

The function was provided the following array of garbage values to store the reconstructed array. Note that it contains enough space to store 20 integers.

```
{1897361756, 511234109, 2040558095, 299669925,
 1854125204, 1147783809, 1808688013, 1528174380,
 1122472434, 999190120, 480027813, 1773684600,
 1495513586, 430636639, 668068513, 1213523779,
 1627837088, 290430805, 479509009, 1697740483, };
```

The function reconstructed the original array as follows, discarding packets #1 and #4 because of the data corruption:

```
{10, 20, 30, 40,
 1854125204, 1147783809, 1808688013, 1528174380,
 90, 100, 110, 120,
 130, 140, 150, 160,
 1627837088, 290430805, 479509009, 1697740483};
```

The return value is `12` because only 12 of the original 18 values could be successfully recovered.

Part 4: Packetize an Array

Complete the `packetize_sf` function, which takes an array of integers in the argument `array` and packetizes the array, i.e., it divides the array into payloads, which the function then attaches to new packets. The function is responsible for allocating memory as needed (via `malloc`) to create each ArrayNet packet.

```
unsigned int packetize_array_sf(int *array, unsigned int array_len,
     unsigned char *packets[], unsigned int packets_len,
     unsigned int max_payload, unsigned int src_addr,
     unsigned int dest_addr, unsigned int src_port,
     unsigned int dest_port, unsigned int maximum_hop_count,
     unsigned int compression_scheme, unsigned int traffic_class)
```

Arguments:

- `array`: the array of signed 32-bit integers to packetize
- `array_len`: the number of elements in array
- `packets`: an array of `char *` pointers. The function must create each packet and store them sequentially in this array so that `packets[0]` points to the first packet, `packets[1]` points to the second packet, etc.
- `packets_len`: the number of elements (pointers) in the `packets` array.
- `max_payload` is the maximum payload size of any packet created and referenced by `packets`.
- `src_addr` and `dest_addr` are the source and destination addresses, respectively, to be stored in each packet created.
- `src_port` and `dest_port` are the source and destination ports, respectively, to be stored in each packet created.
- `maximum_hop_count`: the integer to be stored in the packets which represents the maximum number of hops these packets may take
- `compression_scheme`: the integer to store in the packets which represents the compression algorithm used to compress the payload. Note that you should not attempt to actually compress the data. Just store the `compression_scheme` value as indicated in the header format.

When creating packets, maximize the size of each payload, subject to the `max_payload` constraint. For example, suppose the array to packetize is 20 integers long and `max_payload` is 24 bytes. The function should store 6 integers in each payload of the first three packets, and the remaining 2 integers in the payload of the last packet. The function must also assign correct values to the Fragment Offset, Packet Length and Checksum fields of every packet. Return the number of packets that were stored in `packets`.

As an example of how to use `malloc` to create a packet, suppose `num_bytes` stores the number of bytes you need to allocate for a particular packet and that you want to store that packet at index `i` of the `packets` array. You would write the following line of code:

```
packets[i] = malloc(num_bytes);
```

Then you would write code to set values for all the bytes in the header of the packet and fill in the payload. (These would be the bytes `packets[i][0]`, `packets[i][1]`, ..., `packets[i][num_bytes-1]`)

If `packets` is not long enough to accommodate all the packets needed to packetize the message, fill it with as many packets as possible. Return the number of packets created by the function.

Example

Arguments:

```
int array[] = {17, 89, 42, 631, 52, 77, 89, 100, 125, -6, 823,
     800, 1024, 1025, 9, 1888, 0, -17, 19, 9999999, -888888,
     723, 1000, 1111, -99, -95, 55, };
unsigned int array_len = 27;
unsigned char* packets[4];
unsigned int packets_len = 4;
unsigned int max_payload = 20;
unsigned int src_addr = 93737;
unsigned int dest_addr = 10973;
unsigned int src_port = 11;
unsigned int dest_port = 6;
unsigned int maximum_hop_count = 25;
unsigned int compression_scheme = 3;
unsigned int traffic_class = 14;
```

Packets Generated in `packets[]`:

Packet #0	Packet #1
Source Address: 93737 Destination Address: 10973 Source Port: 11 Destination Port: 6 Fragment Offset: 0 Packet Length: 36 Maximum Hop Count: 25 Checksum: 105636 Compression Scheme: 3 Traffic Class: 14 Payload: 17 89 42 631 52	Source Address: 93737 Destination Address: 10973 Source Port: 11 Destination Port: 6 Fragment Offset: 20 Packet Length: 36 Maximum Hop Count: 25 Checksum: 105222 Compression Scheme: 3 Traffic Class: 14 Payload: 77 89 100 125 -6
Packet #2	**Packet #3**
Source Address: 93737 Destination Address: 10973 Source Port: 11 Destination Port: 6	Source Address: 93737 Destination Address: 10973 Source Port: 11 Destination Port: 6

`Fragment Offset: 40`
`Packet Length: 36`
`Maximum Hop Count: 25`
`Checksum: 108526`
`Compression Scheme: 3`
`Traffic Class: 14`
`Payload: 823 800 1024 1025 9`
	`Fragment Offset: 60`
	`Packet Length: 36`
	`Maximum Hop Count: 25`
	`Checksum: 1718181`
	`Compression Scheme: 3`
	`Traffic Class: 14`
	`Payload: 1888 0 -17 19 9999999`

Return value: `4`

Representations of these Packets in Hexadecimal:

Packet #0
 	
`00 16 e2 90 00 2a dd b6`
`00 00 02 4c 81 9c a4 ce`
`00 00 00 11`
`00 00 00 59`
`00 00 00 2a`
`00 00 02 77`
`00 00 00 34`
	Packet #1
	
	`00 16 e2 90 00 2a dd b6`
	`00 50 02 4c 81 9b 06 ce`
	`00 00 00 4d`
	`00 00 00 59`
	`00 00 00 64`
	`00 00 00 7d`
	`ff ff ff fa`

Packet #2
 	
`00 16 e2 90 00 2a dd b6`
`00 a0 02 4c 81 a7 ee ce`
`00 00 03 37`
`00 00 03 20`
`00 00 04 00`
`00 00 04 01`
`00 00 00 09`
	Packet #3
	
	`00 16 e2 90 00 2a dd b6`
	`00 f0 02 4c 9a 37 a5 ce`
	`00 00 07 60`
	`00 00 00 00`
	`ff ff ff ef`
	`00 00 00 13`
	`00 98 96 7f`

Building and Running Your Code

Two launch targets have been created for you in VS Code:

1. **Run Main**: runs the executable created by compiling the file `hw1_main.c`, which you are welcome to edit as you please.
2. **Run All Tests**: runs the executable `./build/run_all_tests`, which runs a set of unit tests against your implementation in `hw1.c`. Output will be shown in the Terminal tab at the bottom of the VS Code window.

Testing & Grading Notes

- A set of test cases has been provided to you in the template repository. When you start working, the first build will take a while because these many .c files need to be compiled. Subsequent builds will be much shorter.
- It is useful and good practice to create helper functions as you are working on the assignment. These are allowed and will not interfere with grading. However, make sure to only modify `hw1.c`.
- During grading, only your `hw1.c` file will be copied into the grading framework's directory for processing. Make sure all of your code is self-contained in that file.
- Remember to regularly use git to commit and push your work to GitHub. This action will simultaneously submit your work to CodeGrade, where you can see the outcome of additional testing and see your current score on the assignment.
- Your code will be checked both for computational correctness and for correct use of memory via Valgrind. Credit will be awarded for a successful Valgrind test only when most of the computational tests pass.
- Any evidence of hard-coding test case input or output will automatically result in a score of zero for the assignment. This kind of behavior is borderline academic dishonesty and will likely be referred to the University's Academic Judiciary.

Generative AI

Generative AI may not be used to complete this assignment. Evidence of its use will be treated as evidence of academic dishonesty and will be prosecuted as such through the University's Academic Judiciary.

Reminder: Delete Your Codespace When Done!

As discussed earlier in the course, you should delete your Homework #1 Codespace when done so that you don't exceed your monthly GB-month storage quota.

Academic Honesty Policy

Academic honesty is taken very seriously in this course. By submitting your work for grading you indicate your understanding of, and agreement with, the following Academic Honesty Statement:

1. I understand that representing another person's work as my own is academically dishonest.

2. I understand that copying, even with modifications, a solution from another source (such as the web or another person) as a part of my answer constitutes plagiarism.
3. I understand that sharing parts of my homework solutions (algorithms, software design, text write-up, schematics, code, electronic or hard-copy) is academic dishonesty and helps others plagiarize my work.
4. I understand that protecting my work from possible plagiarism is my responsibility. I understand the importance of saving my work such that it is visible only to me.
5. I understand that passing information that is relevant to a homework/exam to others in the course (either lecture or even in the future!) for their private use constitutes academic dishonesty. I will only discuss material that I am willing to openly post on the discussion board.
6. I understand that academic dishonesty is treated very seriously in this course. I understand that the instructor will report any incident of academic dishonesty to the University's Academic Judiciary.
7. I understand that the penalty for academic dishonesty might not be immediately administered. For instance, cheating on a homework assignment may be discovered and penalized after the grades for that homework have been recorded.
8. I understand that buying or paying another entity for any code, partial or in its entirety, and submitting it as my own work is considered academic dishonesty.
9. I understand that there are no extenuating circumstances for academic dishonesty.