CAH11-01 Networks and Operating Systems

Network Access Layer

Aims of the Seminar

Welcome to the Python Workshop on the Network Access Layer! This workshop focuses on the Error Detection & Correction Simulation, Ethernet frame structure, and switching.

Workshop Outline

1) Understand the key concepts of the link layer, including error detection and correction, multiple access protocols (e.g., ALOHA, CSMA/CD), ARP, Ethernet frame structure, and switching.
2) Use Python to simulate network behaviours and analyse performance metrics.
3) Engage with interactive, online platforms to experiment with protocols in real time.

Prerequisites

- Basic Python programming skills
- Understanding of binary numbers and bitwise operations
- Familiarity with networking fundamentals

Feel free to discuss your work with peers, or with any member of the teaching staff.

Reminder

We encourage you to discuss the content of the workshop with the delivery team and any findings you gather from the session.

Workshops are not isolated, if you have questions from previous weeks, or lecture content, please come and talk to us.

Exercises herein represent an example of what to do; feel free to expand upon this.

Helpful Resources

Error Detection
https://www.geeksforgeeks.org/error-detection-in-computer-networks/
Parity check
https://www.geeksforgeeks.org/error-detection-codes-parity-bit-method/
Checksum
https://www.geeksforgeeks.org/error-detection-code-checksum/

Week Recap Quiz

To access the week quiz via:

Week #06: https://learn.gold.ac.uk/mod/quiz/view.php?id=1623714
Week #07: https://learn.gold.ac.uk/mod/quiz/view.php?id=1623926

Exercises

You may find it useful to keep track of your answers from workshops in a separate document, especially for any research tasks.
Where questions are asked of you, this is intended to make you think; it would be wise to write down your responses formally.

GitHub Setup:

Please refer to this introduction for Github:
https://docs.github.com/en/get-started/start-your-journey/hello-world

1. Error Detection & Correction

a. Parity Bits

A **parity bit** is a simple error-detection mechanism used in digital communication and data storage. It is an additional bit appended to a binary message to ensure that the number of 1s in the message follows a predefined rule—either even or odd parity. This technique helps detect single-bit errors that may occur during transmission or storage.

There are two types of parity:

1. **Even parity** – The parity bit is set so that the total number of 1s in the message (including the parity bit) is even.
2. **Odd parity** – The parity bit is set so that the total number of 1s in the message (including the parity bit) is odd.

While parity bits are useful for error detection, they cannot correct errors or detect multiple-bit errors in most cases. They are commonly used in memory systems, serial communication (such as UART), and networking protocols to add a basic level of data integrity.

Original Data	Even Parity	Odd Parity
00000000	0	1
01011011	1	0
01010101	0	1
11111111	0	1
10000000	1	0

Objective: Implement parity checking for 8-bit data.

Activity:

• Write functions that simulate single-bit parity checking and a two-dimensional parity check.

• Simulate error injection (random bit flips) in data frames and verify detection/correction capability.

Code example:

```
# Function to compute even parity bit
def compute_even_parity(data):
    # Sum the data bits and take modulo 2.
    # If sum is odd, parity bit is 1 (to make total even); if even, parity bit
is 0.
    return sum(data) % 2

# Original data bits (example)
data = [1, 0, 1, 0, 1, 1, 0, 0]
parity_bit = compute_even_parity(data)
print("Original Data: ", data)
print("Computed Parity Bit (Even):", parity_bit)

# Transmitted data: append parity bit to the data array
transmitted_data = data + [parity_bit]
print("\nTransmitted Data (Data + Parity):", transmitted_data)
```

Exercise 1: Experiment with different data and review if the check still operate successfully.

Exercise 2: The following script would flip a bit in a predefined location run it and the check still work.

```
# Simulate an error: flip a bit in the transmitted data
# For example, flip the bit at index 3 (0-indexed)
error_index = 3
data_with_error = transmitted_data.copy()
data_with_error[error_index] = 1 - data_with_error[error_index]
print("\nData with an Error Introduced at index", error_index, ":",
data_with_error)

# At the receiver, perform the parity check.
# For even parity, the sum of all bits should be even.
if sum(data_with_error) % 2 == 0:
    print("\nNo error detected (Parity Check Passed)")
else:
    print("\nError detected (Parity Check Failed)")
```

b. 2D Parity bit check

A **2D parity bit** is an extended form of parity checking that enhances error detection by applying parity both **horizontally (row-wise)** and **vertically (column-wise)** in a block of data. This method improves the ability to detect and sometimes even correct certain errors.

How It Works:

1. **Row Parity:** Each row of the data block is assigned a parity bit, ensuring even or odd parity across the row.
2. **Column Parity:** Each column of the data block is assigned a parity bit, ensuring even or odd parity across the column.
3. **Overall Parity:** A final parity bit can be added at the bottom-right corner, covering both row and column parity.

Advantages:

- Detects **multiple-bit errors**, unlike single parity checks.
- In some cases, allows **single-bit error correction** by identifying the exact location of the flipped bit.

2D parity is widely used in applications like RAID (Redundant Array of Independent Disks) storage systems, memory error detection, and network communications to ensure greater data integrity.

Objective: Perform a 2D pasty check
Code Sample:

```
import numpy as np

def compute_parity(mat):
    # Returns (row_parity, col_parity) for even parity
    return np.sum(mat, axis=1) % 2, np.sum(mat, axis=0) % 2

# Create a 4x4 data matrix
data = np.array([
    [1, 0, 1, 1],
    [0, 1, 0, 0],
    [1, 1, 1, 0],
    [0, 0, 1, 1]
])
print("Original Data:\n", data)

# Compute original parity bits
row_par, col_par = compute_parity(data)
print("Row Parity:", row_par)
print("Column Parity:", col_par)

# Introduce an error by flipping the bit at row 2, column 1 (0-indexed)
```

```
data_err = data.copy()
data_err[2, 1] = 1 - data_err[2, 1]
print("\nData with error at (2, 1):\n", data_err)

# Recompute parity bits after error
new_row_par, new_col_par = compute_parity(data_err)
print("New Row Parity:", new_row_par)
print("New Column Parity:", new_col_par)

# Detect and correct the error
err_row = np.where(new_row_par != row_par)[0]
err_col = np.where(new_col_par != col_par)[0]

if err_row.size == 1 and err_col.size == 1:
    error_location = (err_row[0], err_col[0])
    print("\nError detected at:", error_location)
    # Correct the error by flipping the bit back
    data_err[error_location] = 1 - data_err[error_location]
    print("Corrected Data:\n", data_err)
else:
    print("No single-bit error detected or multiple errors occurred.")
```

c. Checksum Using One's Complement

What is a Checksum?

A **checksum** is a method used to detect errors in data transmission or storage. It involves computing a small-sized numerical value (the checksum) from a larger data block and sending it alongside the original data. When the data is received, the receiver re-computes the checksum and compares it with the transmitted checksum to verify data integrity.

Why Use One's Complement for Checksums?

The **one's complement checksum** is a special type of checksum that is widely used in networking, including **Internet Protocol (IP), Transmission Control Protocol (TCP), and User Datagram Protocol (UDP)**. It offers a simple yet effective way to detect errors caused by noise, bit flips, or corruption during transmission.

How the One's Complement Checksum Works

The one's complement checksum operates as follows:

1. **Divide the data into fixed-size words** (typically 16-bit words for networking).
2. **Perform one's complement addition** on all the words.
3. **If a carry occurs**, add it back to the sum (this is called end-around carry).

4. **Take the one's complement** of the final sum to generate the checksum.
5. **Verification:** The receiver adds the checksum to the data sum; if the result is all 1s, the data is valid. Otherwise, an error has occurred.

Example Use Cases of One's Complement Checksum

- **Networking Protocols:** Used in TCP, UDP, and IP header checksums.
- **File Integrity Verification:** Used to verify the correctness of stored or transmitted files.
- **Embedded Systems:** Used for error checking in firmware updates and memory integrity.

The one's complement checksum is efficient and simple, making it a crucial component of error detection mechanisms in digital communication systems.

Code Sample:

```
def ones_complement_sum(a, b, bit_size=16):
    """Perform one's complement addition of two numbers."""
    result = a + b
    if result >= (1 << bit_size):  # If there's an overflow
        result = (result + 1) & ((1 << bit_size) - 1)  # Wrap around carry
    return result

def calculate_checksum(data, bit_size=16):
    """Compute one's complement checksum for a list of integers."""
    checksum = 0
    for word in data:
        checksum = ones_complement_sum(checksum, word, bit_size)
    return ~checksum & ((1 << bit_size) - 1)  # One's complement

def verify_checksum(data, received_checksum, bit_size=16):
    """Verify the checksum by adding it to the computed sum."""
    total = 0
    for word in data:
        total = ones_complement_sum(total, word, bit_size)
    total = ones_complement_sum(total, received_checksum, bit_size)
    return total == (1 << bit_size) - 1  # Valid if all bits are 1

# Example Usage
data = [0b1010101010101010, 0b1100110011001100, 0b1111000011110000]  # Example
16-bit words
checksum = calculate_checksum(data)
print(f"Calculated Checksum: {bin(checksum)}")

# Verification
is_valid = verify_checksum(data, checksum)
print("Checksum is valid" if is_valid else "Checksum is invalid")
```

2. Multiple Access Protocols (MAP)

Objective: Model and compare the performance of Pure ALOHA, Slotted ALOHA, and CSMA/CD.
Activity:
• Develop a simulation that randomly decides when nodes transmit in a time-slotted system.
• Compute the throughput (efficiency) of Slotted ALOHA by varying the transmission probability.
Sample Code for Slotted ALOHA:

```
import random
import matplotlib.pyplot as plt
import numpy as np

def simulate_slotted_aloha(n_nodes, p, n_slots):
    successes = 0
    for _ in range(n_slots):
        # Each node transmits with probability p in a slot
        transmissions = sum(1 for _ in range(n_nodes) if random.random() < p)
        if transmissions == 1:
            successes += 1
    efficiency = successes / n_slots
    return efficiency

n_nodes = 50
ps = np.linspace(0, 1, 50)
efficiencies = [simulate_slotted_aloha(n_nodes, p, 10000) for p in ps]

plt.plot(ps, efficiencies, marker='o')
plt.xlabel('Transmission Probability')
plt.ylabel('Efficiency')
plt.title('Slotted ALOHA Efficiency Simulation')
plt.show()
```

Conclusion

Congratulations! You've completed the Python Workshop on the Network Access Layer. You've learned how to build error correction using Python. Keep experimenting and building more complex applications to deepen your knowledge of networking and Python.

Happy coding! 🚀