Name: Shikha Singh
Roll no: 25
Assignment No.1

Que1. Explain sliding window protocol in detail with the help of a timing diagram?
Ans:

Sliding Window Protocol

Sliding Window Protocol is a method used in data transmission to control the flow of frames between two devices. It ensures efficient, reliable data transmission and is commonly used in data link layer protocols like TCP (Transmission Control Protocol).

The protocol operates by maintaining a "window" that defines the range of frames the sender can transmit before needing an acknowledgment from the receiver. Similarly, the receiver has a window that defines the range of frames it is willing to accept.

This protocol is primarily used to:

- Avoid overwhelming the receiver by sending too much data at once.
- Efficiently utilize network bandwidth by allowing the sender to continue transmitting without waiting for every single acknowledgment.

Key Terms:

1. **Window Size**: The number of frames the sender is allowed to transmit without waiting for an acknowledgment.

2. **Acknowledgment (ACK)**: The receiver sends an acknowledgment to inform the sender that the data has been successfully received.
3. **Sliding**: After receiving an acknowledgment, the window "slides" to allow new frames to be sent or received.
4. **Timeout**: If an acknowledgment is not received within a certain time, the sender retransmits the frame, assuming it was lost.

Types of Sliding Window Protocols:

1. **Go-Back-N (GBN) Protocol**:
    - The sender can send up to **N frames** before receiving an acknowledgment.
    - If an error occurs in any frame, all subsequent frames are retransmitted, starting from the erroneous one.
    - Receiver only accepts frames in order.
2. **Selective Repeat (SR) Protocol**:
    - The sender can send multiple frames, and the receiver can accept them out of order.
    - Only the erroneous frames are retransmitted, improving efficiency.

Working of Sliding Window Protocol

1. **Sender Side**:
    - Sender has a window that allows it to transmit a certain number of frames (equal to window size) without waiting for an acknowledgment.
    - After sending the frames, the sender waits for acknowledgment from the receiver. Once acknowledgment is received, the window slides forward to allow sending new frames.
2. **Receiver Side**:

- The receiver has its own window, which specifies how many frames it can accept.
- It sends back acknowledgments when frames are correctly received.

Example Timing Diagram

Below is a diagram to visualize the sliding window process using **Go-Back-N Protocol**:

```
Time   |  Sender's Window (S)
| Receiver's Window (R)
-----------------------------------------------
---------------------------
t1     | Frame 0, Frame 1, Frame 2, Frame 3
| Waiting for Frame 0
-----------------------------------------------
---------------------------
t2     | Frame 0, Frame 1, Frame 2, Frame 3 sent
| Frame 0 received
        Frame 4
| Acknowledgment 0 sent
-----------------------------------------------
---------------------------
t3     | Sliding window to send Frame 4
| Waiting for Frame 1
-----------------------------------------------
---------------------------
```

```
t4    | Frame 1 received
| Acknowledgment 1 sent
-----------------------------------------------
---------------------------
t5    | Sliding window to send Frame 5
| Waiting for Frame 2
-----------------------------------------------
---------------------------
t6    | Frame 2 received
| Acknowledgment 2 sent
-----------------------------------------------
---------------------------
t7    | Frame 3 lost during transmission
| No acknowledgment sent
-----------------------------------------------
---------------------------
t8    | Timeout: Sender retransmits Frame 3
| Waiting for Frame 3
       Frame 4, Frame 5 withheld
-----------------------------------------------
---------------------------
t9    | Frame 3 received
| Acknowledgment 3 sent
-----------------------------------------------
---------------------------
```

```
t10   | Sliding window to send Frame 6, 7
| Waiting for Frame 4
```

Explanation:

1. **At t1**: The sender sends frames 0, 1, 2, and 3. The window size is 4, so it can send 4 frames before requiring acknowledgment.
2. **At t2**: The receiver successfully receives Frame 0 and sends an acknowledgment (ACK 0).
3. **At t3**: The sender receives ACK 0, slides the window, and is now ready to send Frame 4.
4. **At t7**: Frame 3 is lost, so the receiver does not acknowledge it.
5. **At t8**: The sender does not receive ACK 3, so it times out and retransmits Frame 3.
6. **At t9**: The receiver acknowledges Frame 3, and the sender can now send the next frame.

Advantages of Sliding Window Protocol:

- **Efficient Utilization of Bandwidth**: It allows continuous data transmission, improving efficiency.
- **Flow Control**: It prevents the receiver from being overwhelmed by too many frames at once.
- **Error Control**: With Go-Back-N or Selective Repeat, erroneous frames are identified and retransmitted.

Conclusion

Sliding window protocol is a robust method for ensuring reliable data transmission, particularly in environments where errors are likely. It balances efficiency, error handling, and flow control by allowing

multiple frames to be sent before waiting for acknowledgments while ensuring that frames are retransmitted in case of errors.

Que 2. Difference between the Go-Back-N ARQ and Selective Repeat ARQ?
Ans:

Go-Back-N ARQ (Automatic Repeat reQuest) and **Selective Repeat ARQ** are two types of sliding window protocols used for reliable data transmission. Both protocols ensure that the data is transmitted accurately by retransmitting frames in case of errors. However, they handle errors and retransmissions differently.

Here's a detailed comparison between the two:

1. Basic Concept:

- **Go-Back-N ARQ**: The sender can send multiple frames (up to a specified window size) without waiting for an acknowledgment. However, if an error occurs (or if an acknowledgment is not received for a frame), all frames starting from the erroneous frame must be retransmitted, even if some of them were correctly received.
- **Selective Repeat ARQ**: The sender also sends multiple frames, but in case of an error, only the erroneous frames are retransmitted. The receiver can accept frames out of order and store them in a buffer until the missing frames are received and acknowledged.

2. Window Size:

- **Go-Back-N ARQ**: The window size at the sender is typically **N** frames, but the receiver can only accept one frame (the next expected frame) at a time.
- **Selective Repeat ARQ**: Both the sender and receiver have a window size of **N** frames. The receiver can accept and store multiple frames out of order until the missing frames are received.

3. Handling Errors:

- **Go-Back-N ARQ**: If a frame is lost or an error is detected, all subsequent frames are discarded by the receiver. The sender then retransmits all frames starting from the erroneous one.
- **Selective Repeat ARQ**: If a frame is lost or erroneous, only that specific frame is retransmitted. The receiver accepts and buffers other correctly received frames.

4. Acknowledgment (ACK):

- **Go-Back-N ARQ**: The receiver sends a cumulative acknowledgment (ACK) for the last correctly received and in-sequence frame. If an error occurs, the sender will not receive an acknowledgment for that frame, triggering the retransmission of that frame and all subsequent frames.
- **Selective Repeat ARQ**: The receiver sends individual acknowledgments (ACK) for each frame it successfully receives. This allows for more efficient retransmission, as only the frames with errors need to be resent.

5. Efficiency:

- **Go-Back-N ARQ**: This protocol can be less efficient when errors occur frequently because the sender must retransmit multiple frames, even if some were received correctly.

- **Selective Repeat ARQ**: It is more efficient, especially in error-prone networks, as only erroneous frames are retransmitted, while correct frames are buffered.

6. Buffering at the Receiver:

- **Go-Back-N ARQ**: The receiver does not buffer out-of-order frames. It discards all frames after the erroneous frame until the missing frame is retransmitted and received in the correct sequence.
- **Selective Repeat ARQ**: The receiver buffers out-of-order frames until the missing frame(s) are received. Once the missing frames are correctly received, the buffered frames are processed in order.

7. Complexity:

- **Go-Back-N ARQ**: It is simpler to implement since the receiver does not need to handle out-of-order frames, and there is no need for buffering.
- **Selective Repeat ARQ**: It is more complex to implement because the receiver must maintain a buffer to store out-of-order frames and handle the reordering of frames.

8. Usage of Bandwidth:

- **Go-Back-N ARQ**: It may lead to unnecessary retransmissions, wasting bandwidth, especially when the window size is large and errors occur frequently.
- **Selective Repeat ARQ**: Bandwidth utilization is more efficient since only the erroneous frames are retransmitted, reducing the need for redundant data transmission.

9. Example:

- **Go-Back-N ARQ**: Suppose the sender transmits frames 1, 2, 3, 4, and 5. If frame 3 is lost, the receiver discards frames 4 and 5 and only acknowledges frame 2. The sender will retransmit frames 3, 4, and 5.
- **Selective Repeat ARQ**: If frame 3 is lost, the receiver will accept and buffer frames 4 and 5. Only frame 3 will be retransmitted, and after its reception, the receiver will process frames 3, 4, and 5 in order.

Comparison Table

Feature	Go-Back-N ARQ	Selective Repeat ARQ
Window Size	N frames (sender only)	N frames (both sender and receiver)
Frame Retransmission	Retransmits from the erroneous frame onward	Retransmits only the erroneous frames
Acknowledgment	Cumulative ACK	Individual ACK for each frame
Receiver Buffering	No buffering for out-of-order frames	Buffers out-of-order frames
Error Handling	Retransmits all frames after an error	Only the lost/damaged frame is retransmitted
Implementation Complexity	Simple	More complex

Efficiency	Less efficient in case of errors	More efficient, especially in error-prone networks
Bandwidth Utilization	May waste bandwidth due to unnecessary retransmission	Better bandwidth utilization

Conclusion:

- **Go-Back-N ARQ** is simpler but less efficient in networks where errors occur frequently because it requires retransmission of multiple frames.
- **Selective Repeat ARQ** is more complex but more efficient, as it retransmits only the frames that are lost or corrupted, making it a better choice for reliable communication in error-prone environments.

Que 3. Write a program to implement Kadane's Algorithm?
Ans:

Kadane's Algorithm is an efficient way to find the **maximum sum subarray** in an array of both positive and negative numbers. The algorithm runs in O(n) time complexity, making it optimal for this problem.

Explanation of Kadane's Algorithm:

1. Initialize two variables:
    - `max_so_far` to keep track of the maximum sum encountered so far.

- `max_ending_here` to store the maximum sum of the subarray that ends at the current position.

2. Traverse the array from left to right, updating `max_ending_here` by either adding the current element to it or starting a new subarray at the current element.
3. Update `max_so_far` whenever `max_ending_here` exceeds its value.

Code Implementation of Kadane's Algorithm in C++ :

```
#include <iostream>
#include <climits>
using namespace std;

// Function to implement Kadane's Algorithm
int maxSubArraySum(int arr[], int size) {
    int maxSum = INT_MIN;  // Initialize maximum sum to a very small number
    int currentSum = 0;    // Initialize current sum to 0

    for (int i = 0; i < size; i++) {
        currentSum = currentSum + arr[i];  // Add the current element to current sum

        // If current sum is greater than max sum, update max sum
        if (currentSum > maxSum)
            maxSum = currentSum;

        // If current sum becomes negative, reset it to 0 (start new subarray)
        if (currentSum < 0)
            currentSum = 0;
```

```
    }

    return maxSum;  // Return the maximum subarray sum
}

int main() {
    int arr[] = {-2, 1, -3, 4, -1, 2, 1, -5, 4};
    int size = sizeof(arr) / sizeof(arr[0]);

    // Calling Kadane's Algorithm function to find the maximum subarray
sum
    int maxSum = maxSubArraySum(arr, size);

    // Output the result
    cout << "Maximum Subarray Sum is: " << maxSum << endl;

    return 0;
}
```

Explanation of the Code:

- **`max_so_far`**: Keeps track of the highest sum of any subarray found so far.
- **`max_ending_here`**: Tracks the current subarray sum that ends at the current index.
- The algorithm iterates through the array, adding each element to `max_ending_here`. If at any point `max_ending_here` becomes negative, it is reset to 0, as a negative sum would not contribute to a larger sum subarray.

- At each step, `max_so_far` is updated to store the maximum sum found.

Example:

For the array `{-2, 1, -3, 4, -1, 2, 1, -5, 4}`, Kadane's algorithm will return `6`, which is the sum of the subarray `[4, -1, 2, 1]`.

Time Complexity:

- **Time Complexity**: O(n), where n is the number of elements in the array, because we only traverse the array once.
- **Space Complexity**: O(1), as no extra space is used except for a few variables.

Que 4. What is a prefix sum algorithm?
Ans:

Prefix Sum Algorithm

The **Prefix Sum Algorithm** is a technique used to preprocess an array to make range sum queries (i.e., the sum of elements between two indices) more efficient. It works by creating a new array where each element at index `i` contains the sum of all elements from the beginning of the array up to index `i`.

This approach reduces the time complexity for answering range sum queries from **O(n)** (in a naive approach) to **O(1)**, after an initial **O(n)** preprocessing step to compute the prefix sums.

How Prefix Sum Works:

Let's consider an array `arr` of size `n`:

```
arr = [a0, a1, a2, ..., an-1]
```

The **prefix sum array** `prefixSum` is built as follows:

```
prefixSum[i] = arr[0] + arr[1] + arr[2] + ... + arr[i]
```

Example:

Given an array:

```
arr = [2, 3, -1, 5, 4]
```

The prefix sum array would be:

```
prefixSum = [2, 5, 4, 9, 13]
```

- `prefixSum[0] = arr[0] = 2`
- `prefixSum[1] = arr[0] + arr[1] = 2 + 3 = 5`
- `prefixSum[2] = arr[0] + arr[1] + arr[2] = 2 + 3 - 1 = 4`

- `prefixSum[3] = arr[0] + arr[1] + arr[2] + arr[3] = 2 + 3 - 1 + 5 = 9`
- `prefixSum[4] = arr[0] + arr[1] + arr[2] + arr[3] + arr[4] = 2 + 3 - 1 + 5 + 4 = 13`

Querying Range Sum Using Prefix Sum:

To find the sum of elements between two indices `l` and `r` (inclusive) in the original array, the formula using the prefix sum array is:

```
Sum(l, r) = prefixSum[r] - prefixSum[l-1]
```

If `l == 0`, then the sum is simply `prefixSum[r]`.

Example (continued):

Suppose you want to calculate the sum of elements from index `1` to `3` in the original array:

```
arr[1] + arr[2] + arr[3] = 3 + (-1) + 5 = 7
```

Using the prefix sum array:

```
Sum(1, 3) = prefixSum[3] - prefixSum[0] = 9 - 2 = 7
```

Time Complexity:

- **Preprocessing**: Building the prefix sum array takes **O(n)** time, where n is the size of the array.
- **Range Query**: Once the prefix sum array is built, each range sum query can be answered in **O(1)** time.

Code Implementation in C++:

```
#include <iostream>
using namespace std;

// Function to compute prefix sum array
void computePrefixSum(int arr[], int
prefixSum[], int n) {
    prefixSum[0] = arr[0];
    for (int i = 1; i < n; i++) {
        prefixSum[i] = prefixSum[i - 1] +
arr[i];
    }
}

// Function to get sum of elements from index l
to r
int getRangeSum(int prefixSum[], int l, int r) {
    if (l == 0)
        return prefixSum[r];
    else
        return prefixSum[r] - prefixSum[l - 1];
```

```
}

int main() {
    int arr[] = {2, 3, -1, 5, 4};
    int n = sizeof(arr) / sizeof(arr[0]);

    // Create and compute the prefix sum array
    int prefixSum[n];
    computePrefixSum(arr, prefixSum, n);

    // Query for sum of elements from index 1 to
3
    int l = 1, r = 3;
    cout << "Sum of elements from index " << l
<< " to " << r << " is: "
         << getRangeSum(prefixSum, l, r) <<
endl;

    return 0;
}
```

Output:

```
Sum of elements from index 1 to 3 is: 7
```

Applications of Prefix Sum Algorithm:

1. **Range Sum Queries**: Efficiently answer queries that ask for the sum of elements between two indices.
2. **Subarray Problems**: Useful in solving problems where the sum of subarrays needs to be calculated frequently.
3. **2D Matrix Prefix Sum**: Similar to 1D arrays, prefix sums can be extended to 2D matrices for answering range sum queries within a submatrix.

Conclusion:

The **Prefix Sum Algorithm** is a powerful tool to optimize the performance of range sum queries. It preprocesses the array in linear time and allows range sum queries to be answered in constant time, making it particularly useful in scenarios where multiple queries are made on a static array.

Que 5. Explain pointer technique?
Ans:

The **Pointer Technique** is a powerful concept in programming that involves using pointers to manage and manipulate memory addresses directly. A pointer is a variable that stores the **memory address** of another variable, rather than the variable's value itself. This technique is widely used in languages like C and C++ to achieve dynamic memory management, pass-by-reference, and efficient array and string manipulations.

Key Concepts in Pointer Technique:

Pointer Declaration: A pointer is declared by specifying the type of data it will point to, followed by an asterisk *. For example, to declare a

pointer to an integer:

```
int* ptr;
```

1. Here, `ptr` is a pointer that can hold the address of an `int` variable.

Pointer Initialization: Pointers are typically initialized to the address of an existing variable using the **address-of operator** `&`. For example:

```
int x = 10;
int* ptr = &x;  // ptr now holds the address of
x
```

2.

Dereferencing a Pointer: Dereferencing a pointer means accessing the value stored at the memory address the pointer holds. This is done using the **dereference operator** `*`. For example:

```
int value = *ptr;  // value now holds the value
of x, which is 10
```

3.

Null Pointers: A pointer can also point to nothing by being assigned a special value called `nullptr` in modern C++ (or `NULL` in older C/C++). This is useful to indicate that a pointer does not currently point to a valid address.

```
int* ptr = nullptr;
```

4.
5. **Pointer Arithmetic**: Since pointers store memory addresses, arithmetic operations like increment (`++`), decrement (`--`), addition (`+`), and subtraction (`-`) can be performed on pointers. Pointer arithmetic is based on the size of the data type the pointer points to.
    - For example, incrementing a pointer to an integer (`int`) by 1 will increase its value by `sizeof(int)` (typically 4 bytes):

```
int arr[] = {10, 20, 30};
int* p = arr;
p++;  // p now points to the second element in
arr (arr[1])
```

6.

Passing Pointers to Functions: Pointers can be used to pass variables by reference to functions, allowing the function to modify the original value. This is more efficient than passing large variables by value.

```
void modify(int* ptr) {
    *ptr = 20;  // modifies the value at the
address pointed by ptr
}

int main() {
    int x = 10;
    modify(&x);  // pass address of x
```

```
    cout << x;  // output: 20
}
```

7.

Dynamic Memory Allocation: The pointer technique is essential for dynamic memory management, which allows memory allocation during runtime using functions like `new` and `delete` in C++ (or `malloc` and `free` in C). For example:

```
int* ptr = new int;  // dynamically allocate
memory for an integer
*ptr = 42;           // assign value
delete ptr;          // free memory
```

8.

Example of Pointer Usage:

Here's a simple example illustrating the pointer technique in C++:

```
#include <iostream>
using namespace std;

int main() {
    int var = 10;
    int* ptr = &var;  // Pointer holds the
address of var
```

```
    cout << "Address of var: " << ptr << endl;
// Display address
    cout << "Value of var through pointer: " <<
*ptr << endl;  // Dereference pointer

    *ptr = 20;  // Modify var using pointer
    cout << "New value of var: " << var << endl;
// Output new value

    return 0;
}
```

Output:

```
Address of var: 0x7ffeefbff5c4
Value of var through pointer: 10
New value of var: 20
```

Common Pointer Techniques:

Double Pointers (Pointers to Pointers): A **double pointer** is a pointer that stores the address of another pointer. It's useful when dealing with multidimensional arrays or dynamic memory allocation for structures.

```
int x = 5;
int* ptr = &x;
int** doublePtr = &ptr;
```

```
cout << **doublePtr;  // Outputs 5
```

1.

Array and Pointer Relationship: In C and C++, arrays are closely related to pointers. The name of an array is actually a pointer to the first element of the array.

```
int arr[] = {1, 2, 3};
int* p = arr;  // p points to the first element
of the array
cout << *p;    // Outputs 1
p++;           // Move to the next element
cout << *p;    // Outputs 2
```

2.

Pointer to a Function: A function's address can also be stored in a pointer. This is known as a function pointer and is used in callback mechanisms or event handling.

```
void display() {
    cout << "Hello, World!";
}

int main() {
    void (*funcPtr)() = &display;  // Pointer to
function
    funcPtr();  // Call the function via the
pointer
```

```
    return 0;
}
```

3.

Advantages of the Pointer Technique:

1. **Efficiency**: Pointers allow efficient handling of large data structures by manipulating memory addresses instead of copying values.
2. **Pass by Reference**: Using pointers, functions can modify variables outside their own scope, which is particularly useful in swapping functions and for updating large structures.
3. **Dynamic Memory Management**: Pointers enable the allocation of memory at runtime, which is crucial for creating dynamic data structures like linked lists, trees, etc.
4. **Low-Level Memory Manipulation**: Pointers provide direct access to memory locations, which is useful in system-level programming and resource management.

Disadvantages:

- **Complexity**: Managing pointers can be tricky for beginners, and improper use can lead to errors such as memory leaks, segmentation faults, or undefined behavior.
- **Security Risks**: Improper pointer usage can lead to vulnerabilities like buffer overflows and pointer dereferencing issues.

Conclusion:

The **Pointer Technique** is an essential concept in programming, especially in C and C++. It allows direct memory manipulation, efficient data handling, and dynamic memory allocation, making it a key feature

for performance-critical applications. However, it requires careful management to avoid common pitfalls like memory leaks and dangling pointers.

Que 6. Draw a timing diagram to explain Stop and Wait protocol?
Ans:

The **Stop-and-Wait Protocol** is a fundamental data link layer protocol used for reliable data transmission. In this protocol, the sender transmits one frame at a time and waits for an acknowledgment (ACK) from the receiver before sending the next frame. It is simple but inefficient, especially for long-distance communication, because the sender has to wait for the acknowledgment after sending each frame.

Timing Diagram for Stop-and-Wait Protocol:

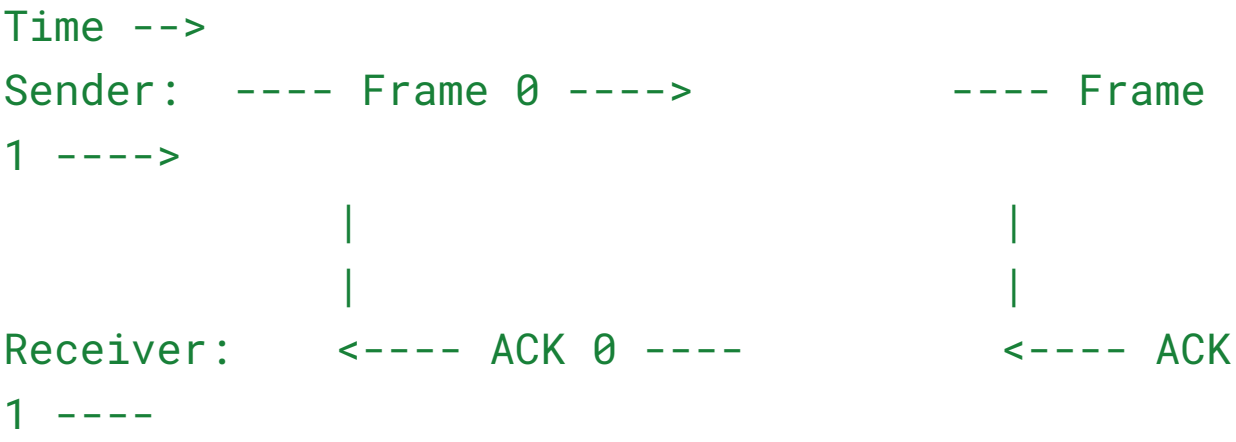
Below is a description of the events in the Stop-and-Wait protocol followed by a timing diagram.

1. **Sender sends Frame 0**: The sender sends a frame (say Frame 0) to the receiver.
2. **Receiver receives Frame 0**: After some propagation delay, the receiver gets the frame.
3. **Receiver sends ACK 0**: The receiver sends an acknowledgment (ACK 0) back to the sender, indicating it successfully received Frame 0.
4. **Sender receives ACK 0**: After some propagation delay, the sender receives the acknowledgment and then sends the next frame (Frame 1).
5. **This process repeats**: The same steps are repeated for each subsequent frame.

Notation:

- **Frame 0, Frame 1**: Frames sent by the sender.
- **ACK 0, ACK 1**: Acknowledgments sent by the receiver.
- **Propagation Delay**: The time it takes for a frame or acknowledgment to travel between the sender and receiver.

Timing Diagram:



Explanation of the Timing Diagram:

- **Frame 0 is sent**: At the beginning of the process, the sender sends Frame 0.
- **Propagation delay**: After a certain time (propagation delay), the frame reaches the receiver.
- **ACK 0 sent by the receiver**: After receiving the frame, the receiver immediately sends back an acknowledgment (ACK 0) to the sender.

- **Propagation delay for ACK**: The acknowledgment takes some time (propagation delay) to reach the sender.
- **Sender receives ACK 0**: The sender gets the acknowledgment for Frame 0 and is now ready to send the next frame, Frame 1.
- **Frame 1 is sent**, and the process repeats: The sender sends the next frame (Frame 1) and waits for ACK 1.

Important Points to Note:

1. **Efficiency**: The Stop-and-Wait protocol is not very efficient for long delays (high propagation delay) or high-speed networks because the sender is idle while waiting for an acknowledgment.
2. **Error Handling**: If a frame or acknowledgment is lost, the sender will eventually timeout and retransmit the frame.
3. **One Frame at a Time**: At any given time, only one frame is in transit between the sender and receiver.

Conclusion:

The **Stop-and-Wait Protocol** is simple but inefficient, as it requires the sender to wait for an acknowledgment before sending the next frame. The timing diagram above illustrates the flow of frames and acknowledgments, along with the associated propagation delays.

Que 7. Find maximum subarray sum for the following elements:

Input: nums = [5,4,-1,7,8]
Input: nums = [1]
Input: nums = [-2,1,-3,4,-1,2,1,-5,4]

Ans:

To find the **maximum subarray sum** for each of the given inputs, we can use **Kadane's Algorithm**, which efficiently solves the problem in **O(n)** time.

Kadane's Algorithm:

- The algorithm iterates through the array and keeps track of the current subarray sum. If the current sum becomes negative, it resets the sum to 0, as a negative sum would reduce the sum of any following subarray.
- At each step, it checks if the current subarray sum is greater than the maximum sum found so far.

Kadane's Algorithm Implementation in C++:

```
#include <iostream>
#include <vector>
#include <climits>
using namespace std;

// Function to implement Kadane's Algorithm
int maxSubArraySum(const vector<int>& nums) {
    int maxSum = INT_MIN;
    int currentSum = 0;

    for (int num : nums) {
        currentSum += num;
        if (currentSum > maxSum)
            maxSum = currentSum;
        if (currentSum < 0)
            currentSum = 0;
    }

    return maxSum;
}

int main() {
    vector<vector<int>> testCases = {
        {5, 4, -1, 7, 8},
        {1},
        {-2, 1, -3, 4, -1, 2, 1, -5, 4}
    };
```

```
    for (const auto& nums : testCases) {
        cout << "Maximum Subarray Sum: " << maxSubArraySum(nums) << endl;
    }

    return 0;
}
```

Explanation:

- The function `maxSubArraySum` computes the maximum subarray sum for a given array using Kadane's Algorithm.
- It iterates through the array, updating `currentSum` and `maxSum` to keep track of the maximum sum found so far.

Output for the Given Inputs:

1. **Input**: `nums = [5, 4, -1, 7, 8]`
    - **Output**: 23
      (The maximum subarray is the entire array `[5, 4, -1, 7, 8]`)
2. **Input**: `nums = [1]`
    - **Output**: 1
      (The maximum subarray is `[1]`)
3. **Input**: `nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]`
    - **Output**: 6
      (The maximum subarray is `[4, -1, 2, 1]`)

Conclusion:

- For `nums = [5, 4, -1, 7, 8]`, the maximum subarray sum is 23.
- For `nums = [1]`, the maximum subarray sum is 1.
- For `nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]`, the maximum subarray sum is 6.